HZ BOOKS
华章科技

自己动手写Java虚拟机

张秀宏 著

Write Your Own Java Virtual Machine

机械工业出版社
China Machine Press

内容简介

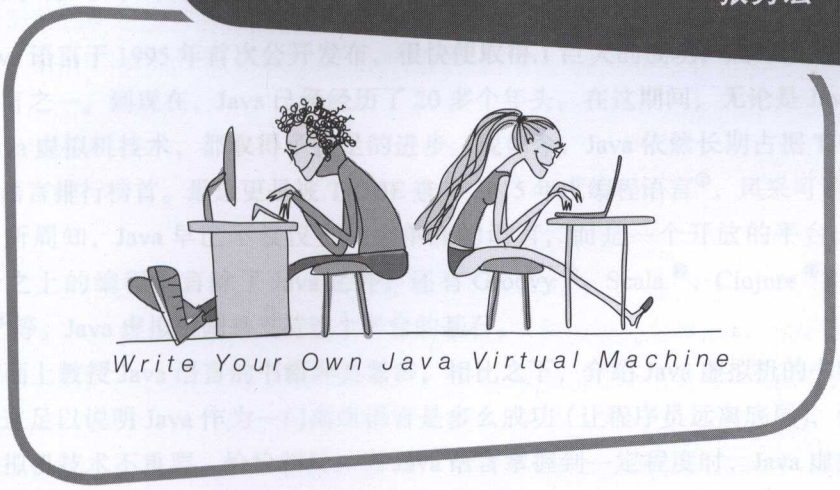
Java虚拟机非常复杂，要想真正理解它的工作原理，最好的方式就是自己动手编写一个！

本书是继《深入理解Java虚拟机》之后的又一经典著作，它一方面遵循《Java虚拟机规范》，一方面又独辟蹊径，不仅能让Java虚拟机的学习变得更加简单和有趣，而且能让你对Java虚拟机的原理认识更深入和更深刻！

本书摒弃了传统的以解读枯燥的Java虚拟机规范文档和分析繁琐的Java虚拟机源代码的方式来讲解Java虚拟机，取而代之的是，以实践的方式，引导读者如何从零开始构建和实现一个Java虚拟机，整个过程不仅能让读者做到对Java虚拟机知其然而且知其所以然，还能屏蔽大量不必要的繁琐细节，体会到实现过程中的成就感，让学习过程更加轻松、愉悦和高效。更重要的是，这种方式能引导读者更深入地认识和掌握Java虚拟机的工作原理。

自己动手写Java虚拟机

张秀宏 著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

自己动手写 Java 虚拟机 / 张秀宏著 . —北京：机械工业出版社，2016.4
（Java 核心技术系列）

ISBN 978-7-111-53413-6

Ⅰ. 自… Ⅱ. 张… Ⅲ. JAVA 语言 – 程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2016）第 066610 号

自己动手写 Java 虚拟机

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：李 艺 责任校对：殷 虹
印 刷：三河市宏图印务有限公司 版 次：2016 年 5 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：17.5
书 号：ISBN 978-7-111-53413-6 定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

Preface 前 言

为什么编写本书

Java语言于1995年首次公开发布，很快便取得了巨大的成功，成为使用最为广泛的编程语言之一。到现在，Java已经经历了20多个年头。在这期间，无论是Java语言本身还是Java虚拟机技术，都取得了长足的进步。现如今，Java依然长期占据TIOBE[㊀]网站的编程语言排行榜首。最近更是被TIOBE选为2015年度编程语言[㊁]，风采可谓不减当年。

众所周知，Java早已不仅仅是一个单纯的语言，而是一个开放的平台。活跃在这个平台之上的编程语言除了Java之外，还有Groovy[㊂]、Scala[㊃]、Clojure[㊄]、Jython[㊅]和JRuby[㊆]等。Java虚拟机则是支持这个平台的基石。

市面上教授Java语言的书籍种类繁多，相比之下，介绍Java虚拟机的书籍却是凤毛麟角。这足以说明Java作为一门高级语言是多么成功（让程序员远离底层），但并不代表Java虚拟机技术不重要。恰恰相反，当Java语言掌握到一定程度时，Java虚拟机原理自然就会成为必须越过的一道鸿沟。

近几年，国内涌现出了一些讨论Java虚拟机技术的优秀书籍，这些书籍主要以分析OpenJDK或Oracle JDK为主。本书另辟蹊径，带领读者自己动手从零开始用Go语言编写Java虚拟机。这样做好处颇多，弥补了OpenJDK等虚拟机的不足。

首先，OpenJDK等虚拟机实现非常复杂。对于初学者而言，很容易陷入代码的海洋

㊀ http://www.tiobe.com/。

㊁ Java曾被TIOBE选为2005年度编程语言。

㊂ http://www.groovy-lang.org/。

㊃ http://www.scala-lang.org/。

㊄ http://clojure.org/。

㊅ http://www.jython.org/。

㊆ http://jruby.org/。

和不必要的细节之中。其次，OpenJDK 等虚拟机大多用 C++ 语言编写。C++ 语言非常复杂，理解起来难度很大。最后，单纯阅读代码比较乏味，缺少乐趣，而脱离代码又很难透彻讨论技术。通过自己动手编写代码，很好地避免了上述问题。看着自己实现的 Java 虚拟机功能逐渐增强，看到可以运行的 Java 程序越来越复杂，成就感非常强。总之，通过实践的方式，相信读者可以更深刻地领悟 Java 虚拟机的工作原理。

Go 是 Google 公司于 2012 年推出的系统编程语言。从到硬件的距离来看，Go 语言介于 C 和 Java 之间。Go 的语法和 C 类似，但更加简洁，因此很容易学习。Go 语言内置了丰富的基本数据类型，并且支持结构体，所以很适合用来实现 Java 虚拟机。Go 支持指针，但并不支持指针运算，因此用 Go 编写的代码要比 C 代码更加安全。此外，Go 还支持垃圾回收和接口等 Java 类语言中才有的功能，大大降低了实现 Java 虚拟机的难度。

以上是本书采用 Go 语言编写 Java 虚拟机的原因，希望读者在学习本书的过程中，可以喜欢上 Go 这门还很年轻的语言。

本书主要内容

全书一共分为 11 章，各章内容安排如下：

第 1 章：安装开发环境，讨论 java 命令，并编写一个类似 Java 的命令行程序。

第 2 章：讨论 Java 虚拟机如何搜索 class 文件，实现类路径。

第 3 章：讨论 class 文件结构，实现 class 文件解析。

第 4 章：讨论运行时数据区，实现线程私有的运行时数据区，包括线程、Java 虚拟机栈、栈帧、操作数栈和局部变量表等。

第 5 章：讨论 Java 虚拟机指令集和解释器，实现解释器和 150 余条指令。

第 6 章：讨论类、对象以及线程共享的运行时数据区，实现类加载器、方法区以及部分引用类指令。

第 7 章：讨论方法调用和返回，实现方法调用和返回指令。

第 8 章：讨论数组和字符串，实现数组相关指令和字符串池。

第 9 章：讨论本地方法调用，实现 Object.hashCode() 等本地方法。

第 10 章：讨论异常处理机制，实现 athrow 指令。

第 11 章：讨论 System 类的初始化过程和 System.out.println() 的工作原理等，并对全书进行总结。

本书面向读者

本书主要面向有一定经验的Java程序员，但任何对Java虚拟机工作原理感兴趣的读者都可以从本书获益。如前所述，本书将使用Go语言实现Java虚拟机。书中会简要介绍Go语言的部分语法以及与Java语言的区别，但不会深入讨论。由于Go语言相对比较简单，相信任何有C系列语言（如C、C++、C#、Objective-C、Java等）经验的读者都可以轻松读懂书中的源代码。

如何阅读本书

本书代码经过精心调整，每一章（第1章除外）都建立在前一章的基础上，但每一章又都可以单独编译和运行。本书内容主要围绕代码对Java虚拟机展开讨论。读者可以从第1章开始，按顺序阅读本书并运行每一章的源代码，也可以直接跳到感兴趣的章节阅读，必要时再阅读其他章节。

参考资料

本书主要参考了下面这些资料：

- 《Java虚拟机规范》第8版
- 《Java语言规范》第8版
- 《深入Java虚拟机》(原书第2版)[⊖]

其中《Java虚拟机规范》主要参考了第8版，但同时也参考了第7版和更老的版本。《Java语言规范》则主要参考了第8版。读者可以从http://docs.oracle.com/javase/specs/index.html获取各个版本的《Java虚拟机规范》和《Java语言规范》。

笔者早在十年前还在上学时就读过由Bill Venners著，曹晓钢等翻译的《深入Java虚拟机（原书第2版）》。但是由于当时水平有限，理解得并不是很深入。时隔十年，重读此书还是颇有收获。较之《Java虚拟机规范》的严谨和刻板，该书更加通俗易懂。原书作者已经将部分章节放于网上，网址是http://www.artima.com/insidejvm/ed2/，读者可以免费阅读。

以上是Java方面的资料。Go语言方面主要参考了Go官网上的各种资料，包括《如

⊖ 原书名为《Inside the Java Virtual Machine，Second Edition》。

何编写 Go 程序》[一]《Effective Go》[二]《Go 语言规范》[三]以及 Go 标准库文档[四]等。另外，在本书的写作过程中，笔者还通过搜索引擎查阅了遍布于网络上（特别是 StackOverflow[五]和 Wikipedia[六]）的各种资料，这里就不一一罗列了。

下载本书源代码

本书源代码可以从 https://github.com/zxh0/jvmgo-book 获取。代码分为 Go 和 Java 两部分，目录结构如下：

```
https://github.com/zxh0/jvmgo-book/v1/code/
|-go
  |-src
    |-jvmgo
|-java
  |-example
```

Go 语言部分是 Java 虚拟机代码，每章为一个子目录，可以独立编译和运行。Java 语言部分是 Java 示例代码，每章为一个包。Java 代码按照 Gradle[七]工程标准目录结构组织，可以用 Gradle 编译整个工程，也可以用 javac 分别编译每个文件。

勘误和支持

《Java 虚拟机规范》对 Java 虚拟机的工作机制有十分严谨的描述。但是由于笔者水平和表达能力有限，本书一定存在表述不精确、不准确，甚至不正确的地方。另外，由于时间有限，书中也难免会有一些疏漏之处，还请读者谅解。

本书的勘误将通过 https://github.com/zxh0/jvmgo-book/blob/master/v1/errata.md 发布和更新。如果读者发现书中的错误、有改进意见，或者有任何问题需要讨论，都可以在本书的 Github 项目上创建 Issue。此外也可以加入 QQ 群（470333113）与读者交流。

[一] https://golang.org/doc/code.html。

[二] https://golang.org/doc/effective_go.html。

[三] https://golang.org/ref/spec。

[四] https://golang.org/pkg/。

[五] http://stackoverflow.com/。

[六] https://en.wikipedia.org/。

[七] http://gradle.org/。

致谢

首先要感谢我的家人和朋友，没有你们的鼓励、支持和帮助，本书不可能面世。这里特别感谢我的妻子，在我陷入低谷的时候，叮嘱我继续努力不要放弃。还有我的朋友范森，每章开头的可爱鼹鼠就是出自他手，希望这些鼹鼠能给枯燥的文字增添一些色彩。

其次感谢我所在的公司乐元素[⊖]，它为我提供了舒适和愉悦的工作环境，使我在工作之余可以全心投入本书的写作之中。

代码被我放到了 Github 上，地址是 https://github.com/zxh0/jvm.go。不过由于能力和时间有限，这个虚拟机离完整实现《 Java 虚拟机规范》还相距甚远。2015 年 4 月份，我停止了 jvm.go 的编写，同时开始改造代码，酝酿本书。感谢所有关注过 jvm.go 项目的人，没有你们的帮助就没有 jvm.go，也就没有本书。

最后，感谢机械工业出版社华章分社的编辑，本书能够顺利出版离不开他们的敬业精神和一丝不苟的工作态度。

⊖ http://www.happyelements.cn/。

目　录 *Contents*

前言

第 1 章　命令行工具……1

1.1　准备工作……1

1.1.1　安装 JDK……1

1.1.2　安装 Go……2

1.1.3　创建目录结构……3

1.2　java 命令……4

1.3　编写命令行工具……5

1.4　测试本章代码……7

1.5　本章小结……8

第 2 章　搜索 class 文件……9

2.1　类路径……9

2.2　准备工作……10

2.3　实现类路径……11

2.3.1　Entry 接口……12

2.3.2　DirEntry……13

2.3.3　ZipEntry……14

2.3.4　CompositeEntry……15

2.3.5　WildcardEntry……17

2.3.6　Classpath……17

2.4　测试本章代码……20

2.5　本章小结……21

第 3 章　解析 class 文件……23

3.1　class 文件……24

3.2　解析 class 文件……25

3.2.1　读取数据……26

3.2.2　整体结构……27

3.2.3　魔数……30

3.2.4　版本号……31

3.2.5　类访问标志……32

3.2.6　类和超类索引……32

3.2.7　接口索引表……33

3.2.8　字段和方法表……33

3.3　解析常量池……35

3.3.1　ConstantPool 结构体……35

3.3.2　ConstantInfo 接口……37

3.3.3　CONSTANT_Integer_info……39

3.3.4　CONSTANT_Float_info……40

3.3.5　CONSTANT_Long_info……40

3.3.6　CONSTANT_Double_info……41

3.3.7　CONSTANT_Utf8_info……42

3.3.8　CONSTANT_String_info……43

3.3.9 CONSTANT_Class_info ····· 45
3.3.10 CONSTANT_NameAnd-Type_info ····· 46
3.3.11 CONSTANT_Fieldref_info、CONSTANT_Methodref_info 和 CONSTANT_Interface-Methodref_info ····· 47
3.3.12 常量池小结 ····· 49
3.4 解析属性表 ····· 50
3.4.1 AttributeInfo 接口 ····· 50
3.4.2 Deprecated 和 Synthetic 属性 ··· 53
3.4.3 SourceFile 属性 ····· 54
3.4.4 ConstantValue 属性 ····· 55
3.4.5 Code 属性 ····· 56
3.4.6 Exceptions 属性 ····· 58
3.4.7 LineNumberTable 和 LocalVariableTable 属性 ····· 59
3.5 测试本章代码 ····· 61
3.6 本章小结 ····· 63

第 4 章 运行时数据区 ····· 65

4.1 运行时数据区概述 ····· 66
4.2 数据类型 ····· 67
4.3 实现运行时数据区 ····· 68
4.3.1 线程 ····· 68
4.3.2 Java 虚拟机栈 ····· 69
4.3.3 帧 ····· 71
4.3.4 局部变量表 ····· 72
4.3.5 操作数栈 ····· 74
4.3.6 局部变量表和操作数栈实例分析 ····· 76
4.4 测试本章代码 ····· 81
4.5 本章小结 ····· 83

第 5 章 指令集和解释器 ····· 85

5.1 字节码和指令集 ····· 86
5.2 指令和指令解码 ····· 88
5.2.1 Instruction 接口 ····· 89
5.2.2 BytecodeReader ····· 91
5.3 常量指令 ····· 92
5.3.1 nop 指令 ····· 92
5.3.2 const 系列指令 ····· 93
5.3.3 bipush 和 sipush 指令 ····· 94
5.4 加载指令 ····· 94
5.5 存储指令 ····· 95
5.6 栈指令 ····· 96
5.6.1 pop 和 pop2 指令 ····· 96
5.6.2 dup 指令 ····· 97
5.6.3 swap 指令 ····· 98
5.7 数学指令 ····· 98
5.7.1 算术指令 ····· 98
5.7.2 位移指令 ····· 99
5.7.3 布尔运算指令 ····· 101
5.7.4 iinc 指令 ····· 102
5.8 类型转换指令 ····· 102
5.9 比较指令 ····· 103
5.9.1 lcmp 指令 ····· 103
5.9.2 fcmp<op> 和 dcmp<op> 指令 ·· 104
5.9.3 if<cond> 指令 ····· 105
5.9.4 if_icmp<cond> 指令 ····· 106
5.9.5 if_acmp<cond> 指令 ····· 107

5.10 控制指令 ····· 108
5.10.1 goto 指令 ····· 108
5.10.2 tableswitch 指令 ····· 108
5.10.3 lookupswitch 指令 ····· 110
5.11 扩展指令 ····· 111
5.11.1 wide 指令 ····· 111
5.11.2 ifnull 和 ifnonnull 指令 ····· 113
5.11.3 goto_w 指令 ····· 113
5.12 解释器 ····· 114
5.13 测试本章代码 ····· 118
5.14 本章小结 ····· 120

第 6 章 类和对象 ····· 121

6.1 方法区 ····· 122
6.1.1 类信息 ····· 122
6.1.2 字段信息 ····· 124
6.1.3 方法信息 ····· 125
6.1.4 其他信息 ····· 127
6.2 运行时常量池 ····· 127
6.2.1 类符号引用 ····· 129
6.2.2 字段符号引用 ····· 130
6.2.3 方法符号引用 ····· 132
6.2.4 接口方法符号引用 ····· 132
6.3 类加载器 ····· 133
6.3.1 readClass() ····· 134
6.3.2 defineClass() ····· 135
6.3.3 link() ····· 136
6.4 对象、实例变量和类变量 ····· 136
6.5 类和字段符号引用解析 ····· 141
6.5.1 类符号引用解析 ····· 141
6.5.2 字段符号引用解析 ····· 142
6.6 类和对象相关指令 ····· 144
6.6.1 new 指令 ····· 144
6.6.2 putstatic 和 getstatic 指令 ····· 146
6.6.3 putfield 和 getfield 指令 ····· 148
6.6.4 instanceof 和 checkcast 指令 ····· 150
6.6.5 ldc 指令 ····· 154
6.7 测试本章代码 ····· 156
6.8 本章小结 ····· 160

第 7 章 方法调用和返回 ····· 161

7.1 方法调用概述 ····· 161
7.2 解析方法符号引用 ····· 163
7.2.1 非接口方法符号引用 ····· 163
7.2.2 接口方法符号引用 ····· 165
7.3 方法调用和参数传递 ····· 166
7.4 返回指令 ····· 169
7.5 方法调用指令 ····· 170
7.5.1 invokestatic 指令 ····· 170
7.5.2 invokespecial 指令 ····· 170
7.5.3 invokevirtual 指令 ····· 172
7.5.4 invokeinterface 指令 ····· 174
7.6 改进解释器 ····· 176
7.7 测试方法调用 ····· 178
7.8 类初始化 ····· 181
7.9 本章小结 ····· 185

第 8 章 数组和字符串 ····· 187

8.1 数组概述 ····· 187
8.2 数组实现 ····· 188
8.2.1 数组对象 ····· 188
8.2.2 数组类 ····· 190

8.2.3 加载数组类……191
8.3 数组相关指令……191
8.3.1 newarray 指令……192
8.3.2 anewarray 指令……194
8.3.3 arraylength 指令……195
8.3.4 <t>aload 指令……196
8.3.5 <t>astore 指令……197
8.3.6 multianewarray 指令……198
8.3.7 完善 instanceof 和 checkcast 指令……201
8.4 测试数组……203
8.5 字符串……204
8.5.1 字符串池……205
8.5.2 完善 ldc 指令……206
8.5.3 完善类加载器……207
8.6 测试字符串……207
8.7 本章小结……210

第 9 章 本地方法调用……211

9.1 注册和查找本地方法……212
9.2 调用本地方法……213
9.3 反射……215
9.3.1 类和对象之间的关系……215
9.3.2 修改类加载器……217
9.3.3 基本类型的类……219
9.3.4 修改 ldc 指令……220
9.3.5 通过反射获取类名……221
9.3.6 测试本节代码……224
9.4 字符串拼接和 String.intern() 方法……225
9.4.1 Java 类库……225
9.4.2 System.arraycopy() 方法……227
9.4.3 Float. floatToRawIntBits() 和 Double.doubleToRawLongBits() 方法……229
9.4.4 String.intern() 方法……229
9.4.5 测试本节代码……230
9.5 Object.hashCode()、equals() 和 toString()……231
9.6 Object.clone()……233
9.7 自动装箱和拆箱……235
9.8 本章小结……238

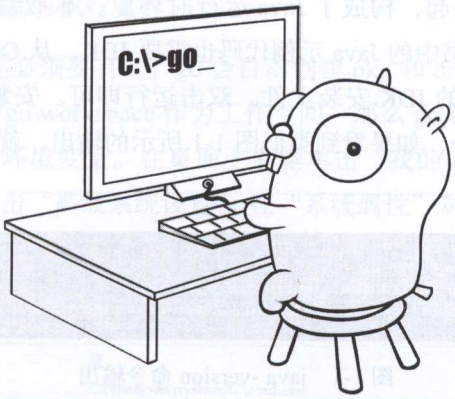
第 10 章 异常处理……239

10.1 异常处理概述……239
10.2 异常抛出……240
10.3 异常处理表……241
10.4 实现 athrow 指令……245
10.5 Java 虚拟机栈信息……248
10.6 测试本章代码……251
10.7 本章小结……252

第 11 章 结束……253

11.1 System 类是如何被初始化的……253
11.2 初始化 System 类……255
11.3 System.out.println() 是如何工作的……258
11.4 测试本章代码……260
11.5 总结……260

附录 指令表……263



第 1 章　命令行工具

Java 虚拟机非常复杂，要想真正理解它的工作原理，最好的方式就是自己动手写一个。本书的目的就是带领读者按照 Java 虚拟机规范[⊖]，从零开始，一步一步用 Go 语言实现一个功能逐步增强的 Java 虚拟机。第 1 章将编写一个类似 java[⊜]的命令行工具，用它来启动我们自己的虚拟机。在开始编写代码之前，需要先准备好开发环境。

本书假定读者使用的是 Windows 操作系统，因此书中出现的命令和路径等都是 Windows 形式的。如果读者使用的是其他操作系统（如 Mac OS X、Linux 等），需要根据自己的情况做出相应调整。由于 Go 和 Java 都是跨平台语言，所以本书代码在常见的操作系统中都可以正常编译和运行。

1.1　准备工作

1.1.1　安装 JDK

我们都知道，要想运行 Java 程序，只有 Java 虚拟机是不够的，还需要有 Java 类库。

⊖ 如无特殊说明，本书中出现的“Java 虚拟机规范”均指《Java 虚拟机规范第 8 版》，网址为 http://docs.oracle.com/javase/specs/jvms/se8/html/index.html。

⊜ 后文中，首字母小写的 java 特指 java 命令行工具。

Java 虚拟机和 Java 类库一起，构成了 Java 运行时环境。本书编写的 Java 虚拟机依赖于 JDK 类库，另外，编译本书中的 Java 示例代码也需要 JDK。从 Oracle 网站[⊖]上下载最新版本（写作本章时是 8u66）的 JDK 安装文件，双击运行即可。安装完毕之后，打开命令行窗口执行 java -version 命令，如果看到类似图 1-1 所示的输出，就证明安装成功了。

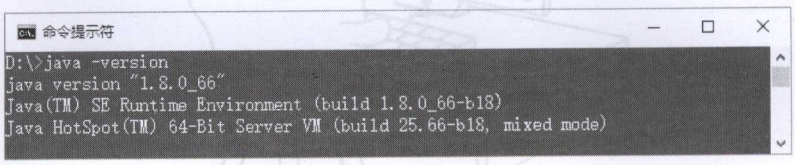


图 1-1　java -version 命令输出

1.1.2　安装 Go

从 Go 语言官网[⊜]下载最新版本（写作本章时是 1.5.1）的 Go 安装文件，双击运行即可。安装完毕之后，打开命令行窗口执行 go version 命令，如果看到类似图 1-2 所示的输出，就证明安装成功了。

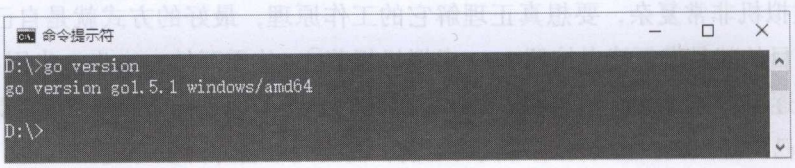


图 1-2　go version 命令输出

go [⊛]命令是 Go 语言提供的命令行工具，用来管理 Go 源代码。go 命令就像瑞士军刀，里面包含了各种小工具。用 Go 语言编写程序，基本上只需要 go 命令就可以了。go 命令里的小工具是各种子命令，version 是其中之一。其他常用的子命令包括 help、fmt、install 和 test 等。

go 命令行工具希望所有的 Go 源代码被都放在一个工作空间中。所谓工作空间，实际上就是一个目录结构，这个目录结构包含三个子目录。

- src 目录中是 Go 语言源代码。
- pkg 目录中是编译好的包对象文件。

⊖ http://www.oracle.com/technetwork/java/javase/downloads/index.html。

⊜ https://golang.org/dl/（如果 Go 官网无法访问，可以从 http://golangtc.com/download）下载。

⊛ 后文中，首字母小写的 go 特指 go 命令行工具。

❑ bin 目录中是链接好的可执行文件。

实际上只有 src 目录是必须要有的，go 会自动创建 pkg 和 bin 目录。工作空间可以位于任何地方，本书使用 D:\go\workspace 作为工作空间。那么 go 如何知道工作空间在哪里呢？答案是通过 GOPATH 环境变量。在桌面上右键单击“我的电脑”图标，在弹出的菜单中单击“属性”，然后单击“高级系统设置”；在“系统属性”对话框中单击“环境变量”按钮，然后添加 GOPATH 变量即可，如图 1-3 所示。

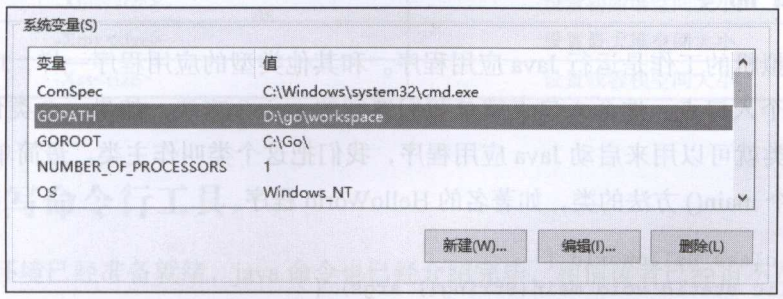


图 1-3　设置 GOPATH 环境变量

打开命令行窗口，执行 go env 命令，如果看到类似图 1-4 所示的输出，GOPATH 环境变量就设置成功了。

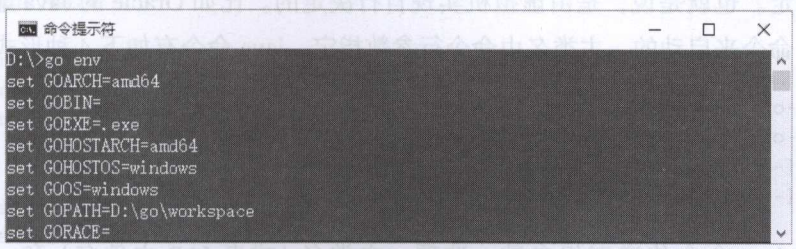


图 1-4　使用 go env 命令查看 GOPATH 环境变量

1.1.3　创建目录结构

Go 语言以包为单位组织源代码，包可以嵌套，形成层次关系。本书编写的 Go 源文件全部放在 jvmgo 包中，其中每一章的源文件又分别放在自己的子包中。包层次和目录结构有一个简单的对应关系，比如，第 1 章的代码在 jvmgo\ch01 目录下。除第 1 章以外，每一章都是先复制前一章代码，然后进行修改和完善。每一章的代码都是独立的，可以单独编译为一个可执行文件。下面创建第 1 章的目录结构。

在 D:\go\workspace\src（也就是 %GOPATH%\src）目录下创建 jvmgo 目录，在 jvmgo 目录下创建 ch01 目录。现在，工作空间的目录结构如下：

```
D:\go\workspace\src
  |-jvmgo
    |-ch01
```

1.2 java 命令

Java 虚拟机的工作是运行 Java 应用程序。和其他类型的应用程序一样，Java 应用程序也需要一个入口点，这个入口点就是我们熟知的 main() 方法。如果一个类包含 main() 方法，这个类就可以用来启动 Java 应用程序，我们把这个类叫作主类。最简单的 Java 程序是只有一个 main() 方法的类，如著名的 HelloWorld 程序。

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

那么 Java 虚拟机如何知道我们要从哪个类启动应用程序呢？对此，Java 虚拟机规范没有明确规定。也就是说，是由虚拟机实现自行决定的。比如 Oracle 的 Java 虚拟机实现是通过 java 命令来启动的，主类名由命令行参数指定。java 命令有如下 4 种形式：

```
java [-options] class [args]
java [-options] -jar jarfile [args]
javaw [-options] class [args]
javaw [-options] -jar jarfile [args]
```

可以向 java 命令传递三组参数：选项、主类名（或者 JAR 文件名）和 main() 方法参数。选项由减号（-）开头。通常，第一个非选项参数给出主类的完全限定名（fully qualified class name）。但是如果用户提供了 -jar 选项，则第一个非选项参数表示 JAR 文件名，java 命令必须从这个 JAR 文件中寻找主类。javaw 命令和 java 命令几乎一样，唯一的差别在于，javaw 命令不显示命令行窗口，因此特别适合用于启动 GUI（图形用户界面）应用程序。

选项可以分为两类：标准选项和非标准选项。标准选项比较稳定，不会轻易变动。非标准选项以 -X 开头，很有可能会在未来的版本中变化。非标准选项中有一部分是高级选

项，以 -XX 开头。表 1-1 列出了 java 命令常用的选项及其用途[⊖]。

表 1-1　java 命令常用选项及其用途

选　项	用　途
-version	输出版本信息，然后退出
-? / -help	输出帮助信息，然后退出
-cp / -classpath	指定用户类路径
-Dproperty=value	设置 Java 系统属性
-Xms<size>	设置初始堆空间大小
-Xmx<size>	设置最大堆空间大小
-Xss<size>	设置线程栈空间大小

1.3　编写命令行工具

开发环境已经准备就绪，java 命令也已经介绍完毕，相信读者已经迫不及待想开始写代码了吧！下面根据 java 命令的第一种用法，自己动手编写一个类似的命令行工具。

先定义一个结构体来表示命令行选项和参数。在 ch01 目录下创建 cmd.go 文件[⊜]，用你喜欢的文本编辑器[⊜]打开它，然后在其中定义 Cmd 结构体，代码如下：

```
package main

import "flag"
import "fmt"
import "os"

type Cmd struct {
    helpFlag    bool
    versionFlag bool
    cpOption    string
    class       string
    args        []string
}
```

在 Java 语言中，API 一般以类库的形式提供。在 Go 语言中，API 则是以包（package）的形式提供。包可以向用户提供常量、变量、结构体以及函数等。Java 内置了丰富的类库，

⊖ 完整的 java 命令用法请参考 http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html。

⊜ Go 源文件一般以 .go 作为后缀，文件名全部小写，多个单词之间用下划线分隔。Go 语言规范要求 Go 源文件必须使用 UTF-8 编码，详见 https://golang.org/ref/spec。

⊜ 笔者推荐 Sublime2，主页为 http://www.sublimetext.com/。

Go 也同样内置了功能强大的包。本章将用到 fmt、os 和 flag 包。

os 包定义了一个 Args 变量，其中存放传递给命令行的全部参数。如果直接处理 os.Args 变量，需要写很多代码。还好 Go 语言内置了 flag 包，这个包可以帮助我们处理命令行选项。有了 flag 包，我们的工作就简单了很多。继续编辑 cmd.go 文件，在其中定义 parseCmd() 函数[⊖]，代码如下：

```
func parseCmd() *Cmd {
    cmd := &Cmd{}

    flag.Usage = printUsage
    flag.BoolVar(&cmd.helpFlag, "help", false, "print help message")
    flag.BoolVar(&cmd.helpFlag, "?", false, "print help message")
    flag.BoolVar(&cmd.versionFlag, "version", false, "print version and exit")
    flag.StringVar(&cmd.cpOption, "classpath", "", "classpath")
    flag.StringVar(&cmd.cpOption, "cp", "", "classpath")
    flag.Parse()

    args := flag.Args()
    if len(args) > 0 {
        cmd.class = args[0]
        cmd.args = args[1:]
    }

    return cmd
}
```

首先设置 flag.Usage 变量，把 printUsage() 函数赋值给它；然后调用 flag 包提供的各种 Var() 函数设置需要解析的选项；接着调用 Parse() 函数解析选项。如果 Parse() 函数解析失败，它就调用 printUsage() 函数把命令的用法打印到控制台。printUsage() 函数的代码如下：

```
func printUsage() {
    fmt.Printf("Usage: %s [-options] class [args...]\n", os.Args[0])
}
```

如果解析成功，调用 flag.Args() 函数可以捕获其他没有被解析的参数。其中第一个参数就是主类名，剩下的是要传递给主类的参数。这样，用了不到 40 行代码，我们的命令行工具就编写完了。下面来测试它。

⊖ Go 语言有函数（Function）和方法（Method）之分，方法调用需要 receiver，函数调用则不需要。

1.4　测试本章代码

在 ch01 目录下创建 main.go 文件，然后输入下面的代码。

```
package main

import "fmt"

func main() {
    cmd := parseCmd()
    if cmd.versionFlag {
        fmt.Println("version 0.0.1")
    } else if cmd.helpFlag || cmd.class == "" {
        printUsage()
    } else {
        startJVM(cmd)
    }
}
```

注意，与 cmd.go 文件一样，main.go 文件的包名也是 main。在 Go 语言中，main 是一个特殊的包，这个包所在的目录（可以叫作任何名字）会被编译为可执行文件。Go 程序的入口也是 main() 函数，但是不接收任何参数，也不能有返回值。

main() 函数先调用 ParseCommand() 函数解析命令行参数，如果一切正常，则调用 startJVM() 函数启动 Java 虚拟机。如果解析出现错误，或者用户输入了 -help 选项，则调用 PrintUsage() 函数打印出帮助信息。如果用户输入了 -version 选项，则输出（一个滥竽充数的）版本信息。因为我们还没有真正开始编写 Java 虚拟机，所以 startJVM() 函数暂时只是打印一些信息而已，代码如下：

```
func startJVM(cmd *Cmd) {
    fmt.Printf("classpath:%s class:%s args:%v\n",
        cmd.cpOption, cmd.class, cmd.args)
}
```

打开命令行窗口，执行下面的命令编译本章代码。

```
go install jvmgo\ch01
```

命令执行完毕后，如果没有看到任何输出就证明编译成功了，此时在 D:\go\workspace\bin 目录下会出现 ch01.exe 文件。现在，可以用各种参数进行测试。笔者的测试结果如图 1-5 所示。

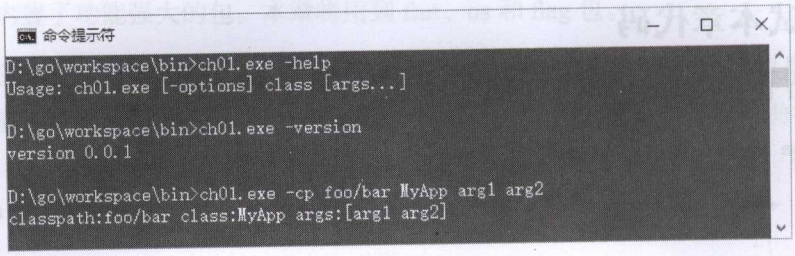
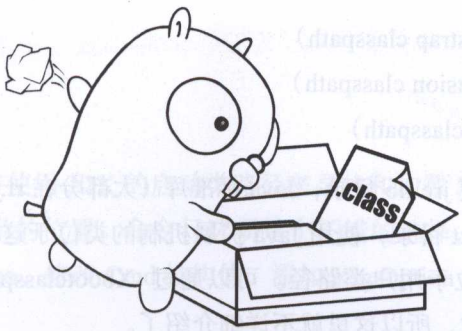


图 1-5 ch01.exe 测试结果

1.5 本章小结

本章准备好了开发环境，学习了 java 命令的基本用法，并且编写了一个简化版的命令行工具。虽然还没有正式开始编写 Java 虚拟机，但是已经打好了坚实的基础。下一章将深入了解 -classpath 选项，探讨 Java 虚拟机从哪里寻找 class 文件，并实现 class 文件加载功能。



第 2 章　搜索 class 文件

第 1 章介绍了 java 命令的用法以及它如何启动 Java 应用程序：首先启动 Java 虚拟机，然后加载主类，最后调用主类的 main() 方法。但是我们知道，即使是最简单的“ Hello, World”程序，也是无法独自运行的，该程序的代码如下：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

加载 HelloWorld 类之前，首先要加载它的超类，也就是 java.lang.Object。在调用 main() 方法之前，因为虚拟机需要准备好参数数组，所以需要加载 java.lang.String 和 java.lang.String[] 类。把字符串打印到控制台还需要加载 java.lang.System 类，等等。那么，Java 虚拟机从哪里寻找这些类呢？本章将详细讨论这个问题。

2.1　类路径

Java 虚拟机规范并没有规定虚拟机应该从哪里寻找类，因此不同的虚拟机实现可以采用不同的方法。Oracle 的 Java 虚拟机实现根据类路径（class path）来搜索类。按照搜索的

先后顺序，类路径可以分为以下 3 个部分：

- 启动类路径（bootstrap classpath）
- 扩展类路径（extension classpath）
- 用户类路径（user classpath）

启动类路径默认对应 jre\lib 目录，Java 标准库（大部分在 rt.jar 里）位于该路径。扩展类路径默认对应 jre\lib\ext 目录，使用 Java 扩展机制的类位于这个路径。我们自己实现的类，以及第三方类库则位于用户类路径。可以通过 -Xbootclasspath 选项修改启动类路径，不过通常并不需要这样做，所以这里就不详细介绍了。

用户类路径的默认值是当前目录，也就是“.”。可以设置 CLASSPATH 环境变量来修改用户类路径，但是这样做不够灵活，所以不推荐使用。更好的办法是给 java 命令传递 -classpath（或简写为 -cp）选项。-classpath/-cp 选项的优先级更高，可以覆盖 CLASSPATH 环境变量设置。第 1 章简单介绍过这个选项，这里再详细解释一下。

-classpath/-cp 选项既可以指定目录，也可以指定 JAR 文件或者 ZIP 文件，如下：

```
java -cp path\to\classes ...
java -cp path\to\lib1.jar ...
java -cp path\to\lib2.zip ...
```

还可以同时指定多个目录或文件，用分隔符分开即可。分隔符因操作系统而异。在 Windows 系统下是分号，在类 UNIX（包括 Linux、Mac OS X 等）系统下是冒号。例如在 Windows 下：

```
java -cp path\to\classes;lib\a.jar;lib\b.jar;lib\c.zip ...
```

从 Java 6 开始，还可以使用通配符（*）指定某个目录下的所有 JAR 文件，格式如下：

```
java -cp classes;lib\* ...
```

2.2 准备工作

从第 2 章开始，每章的代码都是建立在前一章的基础之上。把 ch01 目录复制一份，然后改名为 ch02。因为本章要创建的源文件都在 classpath 包中，所以在 ch02 目录中创建一个 classpath 子目录。现在目录结构看起来应该是这样：

```
D:\go\workspace\src
```

```
|-jvmgo
  |-ch01
  |-ch02
    |-classpath
    |-cmd.go
    |-main.go
```

我们的 Java 虚拟机将使用 JDK 的启动类路径来寻找和加载 Java 标准库中的类，因此需要某种方式指定 jre 目录的位置。命令行选项是个不错的选择，所以增加一个非标准选项 -Xjre。打开 ch02\cmd.go，修改 Cmd 结构体，添加 XjreOption 字段，代码如下：

```
type Cmd struct {
    helpFlag        bool
    versionFlag     bool
    cpOption        string
    XjreOption      string
    class           string
    args            []string
}
```

parseCmd() 函数也要相应修改，代码如下：

```
func parseCmd() *Cmd {
    cmd := &Cmd{}
    flag.Usage = printUsage
    flag.BoolVar(&cmd.helpFlag, "help", false, "print help message")
    flag.BoolVar(&cmd.helpFlag, "?", false, "print help message")
    flag.BoolVar(&cmd.versionFlag, "version", false, "print version and exit")
    flag.StringVar(&cmd.cpOption, "classpath", "", "classpath")
    flag.StringVar(&cmd.cpOption, "cp", "", "classpath")
    flag.StringVar(&cmd.XjreOption, "Xjre", "", "path to jre")
    flag.Parse()
    ... // 其他代码不变
}
```

2.3　实现类路径

可以把类路径想象成一个大的整体，它由启动类路径、扩展类路径和用户类路径三个小路径构成。三个小路径又分别由更小的路径构成。是不是很像组合模式（composite pattern）？没错，本节就套用组合模式来设计和实现类路径。

2.3.1 Entry 接口

先定义一个接口来表示类路径项。在 ch02\classpath 目录下创建 entry.go 文件，在其中定义 Entry 接口，代码如下：

```
package classpath

import "os"
import "strings"

const pathListSeparator = string(os.PathListSeparator)

type Entry interface {
    readClass(className string) ([]byte, Entry, error)
    String() string
}

func newEntry(path string) Entry {...}
```

常量 pathListSeparator 是 string 类型，存放路径分隔符，后面会用到。Entry 接口中有两个方法。readClass() 方法负责寻找和加载 class 文件；String() 方法的作用相当于 Java 中的 toString()，用于返回变量的字符串表示。

readClass() 方法的参数是 class 文件的相对路径，路径之间用斜线（/）分隔，文件名有 .class 后缀。比如要读取 java.lang.Object 类，传入的参数应该是 java/lang/Object.class。返回值是读取到的字节数据、最终定位到 class 文件的 Entry，以及错误信息。Go 的函数或方法允许返回多个值，按照惯例，可以使用最后一个返回值作为错误信息。

newEntry() 函数根据参数创建不同类型的 Entry 实例，代码如下：

```
func newEntry(path string) Entry {
    if strings.Contains(path, pathListSeparator) {
        return newCompositeEntry(path)
    }
    if strings.HasSuffix(path, "*") {
        return newWildcardEntry(path)
    }
    if strings.HasSuffix(path, ".jar") || strings.HasSuffix(path, ".JAR") ||
        strings.HasSuffix(path, ".zip") || strings.HasSuffix(path, ".ZIP") {

        return newZipEntry(path)
    }

    return newDirEntry(path)
}
```

Entry 接口有 4 个实现，分别是 DirEntry、ZipEntry、CompositeEntry 和 WildcardEntry。下面分别介绍每一种实现。

2.3.2　DirEntry

在 4 种实现中，DirEntry 相对简单一些，表示目录形式的类路径。在 ch02\classpath 目录下创建 entry_dir.go 文件，在其中定义 DirEntry 结构体，代码如下：

```
package classpath

import "io/ioutil"
import "path/filepath"

type DirEntry struct {
    absDir string
}

func newDirEntry(path string) *DirEntry {...}
func (self *DirEntry) readClass(className string) ([]byte , Entry, error) {...}
func (self *DirEntry) String() string {...}
```

DirEntry 只有一个字段，用于存放目录的绝对路径。和 Java 语言不同，Go 结构体不需要显示实现接口，只要方法匹配即可。Go 没有专门的构造函数，本书统一使用 new 开头的函数来创建结构体实例，并把这类函数称为构造函数。newDirEntry() 函数的代码如下：

```
func newDirEntry(path string) *DirEntry {
    absDir, err := filepath.Abs(path)
    if err != nil {
        panic(err)
    }
    return &DirEntry{absDir}
}
```

newDirEntry() 先把参数转换成绝对路径，如果转换过程出现错误，则调用 panic() 函数终止程序执行，否则创建 DirEntry 实例并返回。

下面介绍 readClass() 方法：

```
func (self *DirEntry) readClass(className string) ([]byte, Entry, error) {
    fileName := filepath.Join(self.absDir, className)
    data, err := ioutil.ReadFile(fileName)
    return data, self, err
}
```

readClass() 先把目录和 class 文件名拼成一个完整的路径，然后调用 ioutil 包提供的 ReadFile() 函数读取 class 文件内容，最后返回。String() 方法很简单，直接返回目录，代码如下：

```
func (self *DirEntry) String() string {
    return self.absDir
}
```

2.3.3 ZipEntry

ZipEntry 表示 ZIP 或 JAR 文件形式的类路径。在 ch02\classpath 目录下创建 entry_zip.go 文件，在其中定义 ZipEntry 结构体，代码如下：

```
package classpath

import "archive/zip"
import "errors"
import "io/ioutil"
import "path/filepath"

type ZipEntry struct {
    absPath string
}

func newZipEntry(path string) *ZipEntry {...}
func (self *ZipEntry) readClass(className string) ([]byte, Entry, error) {...}
func (self *ZipEntry) String() string {...}
```

absPath 字段存放 ZIP 或 JAR 文件的绝对路径。构造函数和 String() 与 DirEntry 大同小异，就不多解释了，代码如下：

```
func newZipEntry(path string) *ZipEntry {
    absPath, err := filepath.Abs(path)
    if err != nil {
        panic(err)
    }
    return &ZipEntry{absPath}
}

func (self *ZipEntry) String() string {
    return self.absPath
}
```

下面重点介绍如何从 ZIP 文件中提取 class 文件，代码如下：

```
func (self *ZipEntry) readClass(className string) ([]byte, Entry, error) {
    r, err := zip.OpenReader(self.absPath)
    if err != nil {
        return nil, nil, err
    }

    defer r.Close()
    for _, f := range r.File {
        if f.Name == className {
            rc, err := f.Open()
            if err != nil {
                return nil, nil, err
            }

            defer rc.Close()
            data, err := ioutil.ReadAll(rc)
            if err != nil {
                return nil, nil, err
            }

            return data, self, nil
        }
    }

    return nil, nil, errors.New("class not found: " + className)
}
```

首先打开 ZIP 文件，如果这一步出错的话，直接返回。然后遍历 ZIP 压缩包里的文件，看能否找到 class 文件。如果能找到，则打开 class 文件，把内容读取出来，并返回。如果找不到，或者出现其他错误，则返回错误信息。有两处使用了 defer 语句来确保打开的文件得以关闭。readClass() 方法每次都要打开和关闭 ZIP 文件，因此效率不是很高。笔者进行了优化，但鉴于篇幅有限，就不展示具体代码了。感兴趣的读者可以阅读 ch02\classpath\entry_zip2.go 文件。

2.3.4　CompositeEntry

在 ch02\classpath 目录下创建 entry_composite.go 文件，在其中定义 CompositeEntry 结构体，代码如下：

```
package classpath

import "errors"
import "strings"
```

```
type CompositeEntry []Entry

func newCompositeEntry(pathList string) CompositeEntry {...}
func (self CompositeEntry) readClass(className string) ([]byte, Entry, error) {...}
func (self CompositeEntry) String() string {...}
```

如前所述，CompositeEntry 由更小的 Entry 组成，正好可以表示成 []Entry。在 Go 语言中，数组属于比较低层的数据结构，很少直接使用。大部分情况下，使用更便利的 slice 类型。构造函数把参数（路径列表）按分隔符分成小路径，然后把每个小路径都转换成具体的 Entry 实例，代码如下：

```
func newCompositeEntry(pathList string) CompositeEntry {
    compositeEntry := []Entry{}
    for _, path := range strings.Split(pathList, pathListSeparator) {
        entry := newEntry(path)
        compositeEntry = append(compositeEntry, entry)
    }
    return compositeEntry
}
```

相信读者已经想到 readClass() 方法的代码了：依次调用每一个子路径的 readClass() 方法，如果成功读取到 class 数据，返回数据即可；如果收到错误信息，则继续；如果遍历完所有的子路径还没有找到 class 文件，则返回错误。readClass() 方法的代码如下：

```
func (self CompositeEntry) readClass(className string) ([]byte, Entry, error) {
    for _, entry := range self {
        data, from, err := entry.readClass(className)
        if err == nil {
            return data, from, nil
        }
    }
    return nil, nil, errors.New("class not found: " + className)
}
```

String() 方法也不复杂。调用每一个子路径的 String() 方法，然后把得到的字符串用路径分隔符拼接起来即可，代码如下：

```
func (self CompositeEntry) String() string {
    strs := make([]string, len(self))
    for i, entry := range self {
        strs[i] = entry.String()
    }
    return strings.Join(strs, pathListSeparator)
}
```

2.3.5　WildcardEntry

WildcardEntry 实际上也是 CompositeEntry，所以就不再定义新的类型了。在 ch02\classpath 目录下创建 entry_wildcard.go 文件，在其中定义 newWildcardEntry() 函数，代码如下：

```
package classpath

import "os"
import "path/filepath"
import "strings"

func newWildcardEntry(path string) CompositeEntry {
    baseDir := path[:len(path)-1] // remove *
    compositeEntry := []Entry{}

    walkFn := func(path string, info os.FileInfo, err error) error {...}
    filepath.Walk(baseDir, walkFn)

    return compositeEntry
}
```

首先把路径末尾的星号去掉，得到 baseDir，然后调用 filepath 包的 Walk() 函数遍历 baseDir 创建 ZipEntry。Walk() 函数的第二个参数也是一个函数，了解函数式编程的读者应该一眼就可以认出这种用法（即函数可作为参数）。walkFn 变量的定义如下：

```
walkFn := func(path string, info os.FileInfo, err error) error {
    if err != nil {
        return err
    }
    if info.IsDir() && path != baseDir {
        return filepath.SkipDir
    }
    if strings.HasSuffix(path, ".jar") || strings.HasSuffix(path, ".JAR") {
        jarEntry := newZipEntry(path)
        compositeEntry = append(compositeEntry, jarEntry)
    }
    return nil
}
```

在 walkFn 中，根据后缀名选出 JAR 文件，并且返回 SkipDir 跳过子目录（通配符类路径不能递归匹配子目录下的 JAR 文件）。

2.3.6　Classpath

Entry 接口和 4 个实现介绍完了，接下来实现 Classpath 结构体。还是在 ch02\classpath

目录下创建 classpath.go 文件，把下面的代码输入进去。

```
package classpath

import "os"
import "path/filepath"

type Classpath struct {
    bootClasspath Entry
    extClasspath  Entry
    userClasspath Entry
}

func Parse(jreOption, cpOption string) *Classpath {...}
func (self *Classpath) ReadClass(className string) ([]byte, Entry, error) {...}
func (self *Classpath) String() string {...}
```

Classpath 结构体有三个字段，分别存放三种类路径。Parse() 函数使用 -Xjre 选项解析启动类路径和扩展类路径，使用 -classpath/-cp 选项解析用户类路径，代码如下：

```
func Parse(jreOption, cpOption string) *Classpath {
    cp := &Classpath{}
    cp.parseBootAndExtClasspath(jreOption)
    cp.parseUserClasspath(cpOption)
    return cp
}
```

parseBootAndExtClasspath() 方法的代码如下：

```
func (self *Classpath) parseBootAndExtClasspath(jreOption string) {
    jreDir := getJreDir(jreOption)

    // jre/lib/*
    jreLibPath := filepath.Join(jreDir, "lib", "*")
    self.bootClasspath = newWildcardEntry(jreLibPath)

    // jre/lib/ext/*
    jreExtPath := filepath.Join(jreDir, "lib", "ext", "*")
    self.extClasspath = newWildcardEntry(jreExtPath)
}
```

优先使用用户输入的 -Xjre 选项作为 jre 目录。如果没有输入该选项，则在当前目录下寻找 jre 目录。如果找不到，尝试使用 JAVA_HOME 环境变量。getJreDir() 函数的代码如下：

```
func getJreDir(jreOption string) string {
    if jreOption != "" && exists(jreOption) {
        return jreOption
```

```
    }
    if exists("./jre") {
        return "./jre"
    }
    if jh := os.Getenv("JAVA_HOME"); jh != "" {
        return filepath.Join(jh, "jre")
    }
    panic("Can not find jre folder!")
}
```

exists() 函数用于判断目录是否存在，代码如下：

```
func exists(path string) bool {
    if _, err := os.Stat(path); err != nil {
        if os.IsNotExist(err) {
            return false
        }
    }
    return true
}
```

parseUserClasspath() 方法的代码相对简单一些，如下：

```
func (self *Classpath) parseUserClasspath(cpOption string) {
    if cpOption == "" {
        cpOption = "."
    }
    self.userClasspath = newEntry(cpOption)
}
```

如果用户没有提供 -classpath/-cp 选项，则使用当前目录作为用户类路径。ReadClass() 方法依次从启动类路径、扩展类路径和用户类路径中搜索 class 文件，代码如下：

```
func (self *Classpath) ReadClass(className string) ([]byte, Entry, error) {
    className = className + ".class"
    if data, entry, err := self.bootClasspath.readClass(className); err == nil {
        return data, entry, err
    }
    if data, entry, err := self.extClasspath.readClass(className); err == nil {
        return data, entry, err
    }
    return self.userClasspath.readClass(className)
}
```

注意，传递给 ReadClass() 方法的类名不包含“.class”后缀。最后，String() 方法返回用户类路径的字符串表示，代码如下：

```
func (self *Classpath) String() string {
    return self.userClasspath.String()
}
```

至此，整个类路径都实现了，下面我们来测试一下。

2.4 测试本章代码

打开 ch02/main.go 文件，添加两条 import 语句，代码如下：

```
package main

import "fmt"
import "strings"
import "jvmgo/ch02/classpath"

func main() {...}
func startJVM(cmd *Cmd) {...}
```

main() 函数不用变，重写 startJVM() 函数，代码如下：

```
func startJVM(cmd *Cmd) {
    cp := classpath.Parse(cmd.XjreOption, cmd.cpOption)
    fmt.Printf("classpath:%v class:%v args:%v\n",
        cp, cmd.class, cmd.args)

    className := strings.Replace(cmd.class, ".", "/", -1)
    classData, _, err := cp.ReadClass(className)
    if err != nil {
        fmt.Printf("Could not find or load main class %s\n", cmd.class)
        return
    }

    fmt.Printf("class data:%v\n", classData)
}
```

startJVM() 先打印出命令行参数，然后读取主类数据，并打印到控制台。虽然还是无法真正启动 Java 虚拟机，不过相比第 1 章，已经有了很大的进步。打开命令行窗口，执行下面的命令编译本章代码。

```
go install jvmgo\ch02
```

编译成功后，在 D:\go\workspace\bin 目录下出现 ch02.exe 文件。执行 ch02.exe，指定好 -Xjre 选项和类名，就可以把 class 文件的内容打印出来。虽然只是一堆看似杂乱无章的

数字，但成就感还是会油然而生。笔者的测试结果如图 2-1 所示。

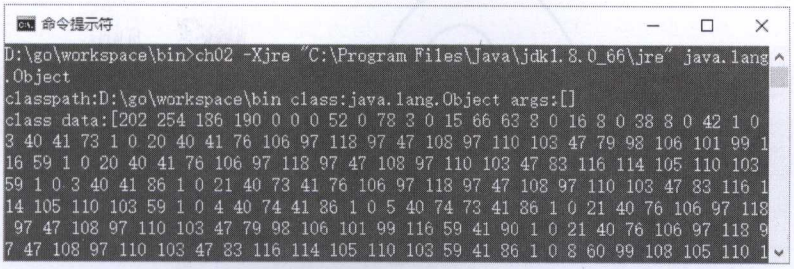
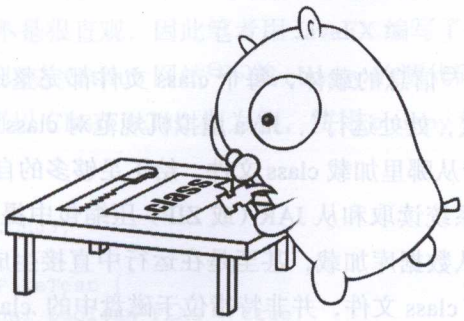


图 2-1　ch02.exe 的测试结果

2.5　本章小结

本章讨论了 Java 虚拟机从哪里寻找 class 文件，对类路径和 -classpath 命令行选项有了较为深入的了解，并且把抽象的类路径概念转变成了具体的代码。下一章将研究 class 文件格式，实现 class 文件解析。



第 3 章　解析 class 文件

第 2 章介绍了 Java 虚拟机从哪里搜索 class 文件，并且实现了类路径功能，已经可以把 class 文件读取到内存中。本章将详细讨论 class 文件格式，编写代码解析 class 文件，为下一步真正实现 Java 虚拟机做好准备。

在开始阅读本章之前，先把目录结构准备好。复制 ch02 目录，并改名为 ch03，然后编辑 ch03\main.go 等文件，把 import 语句中的 ch02 都改成 ch03 [⊖]，最后在 ch03 目录中创建 classfile 子目录。现在的目录结构看起来应该如下所示：

```
D:\go\workspace\src
  |-jvmgo
    |-ch01
    |-ch02
    |-ch03
      |-classfile
      |-classpath
      |-cmd.go
      |-main.go
```

⊖ 这个过程比较无趣，也容易出错。可以使用编辑器提供的“搜索和替换”功能来完成这项工作。

3.1 class 文件

作为类（或者接口）[⊖]信息的载体，每个 class 文件都完整地定义了一个类。为了使 Java 程序可以“编写一次，处处运行”，Java 虚拟机规范对 class 文件格式进行了严格的规定。但是另一方面，对于从哪里加载 class 文件，给了足够多的自由。由第 2 章可知，Java 虚拟机实现可以从文件系统读取和从 JAR（或 ZIP）压缩包中提取 class 文件。除此之外，也可以通过网络下载、从数据库加载，甚至是在运行中直接生成 class 文件。Java 虚拟机规范（和本书）中所指的 class 文件，并非特指位于磁盘中的 .class 文件，而是泛指任何格式符合规范的 class 数据。

构成 class 文件的基本数据单位是字节，可以把整个 class 文件当成一个字节流来处理。稍大一些的数据由连续多个字节构成，这些数据在 class 文件中以大端（big-endian）方式存储。为了描述 class 文件格式，Java 虚拟机规范定义了 u1、u2 和 u4 三种数据类型来表示 1、2 和 4 字节无符号整数，分别对应 Go 语言的 uint8、uint16 和 uint32 类型。相同类型的多条数据一般按表（table）的形式存储在 class 文件中。表由表头和表项（item）构成，表头是 u2 或 u4 整数。假设表头是 n，后面就紧跟着 n 个表项数据。

Java 虚拟机规范使用一种类似 C 语言的结构体语法来描述 class 文件格式。整个 class 文件被描述为一个 ClassFile 结构，代码如下：

```
ClassFile {
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info         constant_pool[constant_pool_count-1];
    u2              access_flags;
    u2              this_class;
    u2              super_class;
    u2              interfaces_count;
    u2              interfaces[interfaces_count];
    u2              fields_count;
    field_info      fields[fields_count];
    u2              methods_count;
    method_info     methods[methods_count];
    u2              attributes_count;
    attribute_info  attributes[attributes_count];
}
```

⊖ 本章后面提到的类，如无特别说明，均泛指类或者接口。

JDK 提供了一个功能强大的命令行工具 javap，可以用它反编译 class 文件。不过从控制台观察 javap 的输出并不是很直观，因此笔者用 JavaFX 编写了一个图形化的工具，叫作 classpy。有兴趣的读者可以去 GitHub 网站[⊖]下载 classpy 的源代码或者打包好的 JAR 可执行文件。后面的小节中将以 ClassFileTest 类为例，使用 classpy 程序分析 class 文件格式。ClassFileTest 的代码如下：

```
package jvmgo.book.ch03;

public class ClassFileTest {
    public static final boolean FLAG = true;
    public static final byte BYTE = 123;
    public static final char X = 'X';
    public static final short SHORT = 12345;
    public static final int INT = 123456789;
    public static final long LONG = 12345678901L;
    public static final float PI = 3.14f;
    public static final double E = 2.71828;

    public static void main(String[] args) throws RuntimeException {
        System.out.println("Hello, World!");
    }
}
```

3.2　解析 class 文件

本节将一边讨论 class 文件格式，一边编写代码实现 class 文件解析。Go 语言内置了丰富的数据类型，非常适合处理 class 文件。为了便于读者参考，表 3-1 给出了 Go 和 Java 语言基本数据类型对照关系。在第 4 章中还会继续讨论 Java 数据类型。

表 3-1　Go 和 Java 语言基本数据类型对照关系

Go 语言类型	Java 语言类型	说　明
int8	byte	8 比特有符号整数
uint8（别名 byte）	N/A	8 比特无符号整数
int16	short	16 比特有符号整数
uint16	char	16 比特无符号整数
int32（别名 rune）	int	32 比特有符号整数
uint32	N/A	32 比特无符号整数
int64	long	64 比特有符号整数

⊖ https://github.com/zxh0/classpy。

（续）

Go 语言类型	Java 语言类型	说　明
uint64	N/A	64 比特无符号整数
float33	float	32 比特 IEEE-754 浮点数
float64	double	64 比特 IEEE-754 浮点数

3.2.1 读取数据

解析 class 文件的第一步是从里面读取数据。虽然可以把 class 文件当成字节流来处理，但是直接操作字节很不方便，所以先定义一个结构体来帮助读取数据。在 ch03\classfile 目录下创建 class_reader.go 文件，在其中定义 ClassReader 结构体和数据读取方法，代码如下：

```
package classfile

import "encoding/binary"

type ClassReader struct {
    data []byte
}

func (self *ClassReader) readUint8() uint8 {...} // u1
func (self *ClassReader) readUint16() uint16 {...} // u2
func (self *ClassReader) readUint32() uint32 {...} // u4
func (self *ClassReader) readUint64() uint64 {...}
func (self *ClassReader) readUint16s() []uint16 {...}
func (self *ClassReader) readBytes(length uint32) []byte {...}
```

ClassReader 只是 []byte 类型的包装而已。readUint8() 读取 u1 类型数据，代码如下：

```
func (self *ClassReader) readUint8() uint8 {
   val := self.data[0]
   self.data = self.data[1:]
   return val
}
```

注意，ClassReader 并没有使用索引记录数据位置，而是使用 Go 语言的 reslice 语法跳过已经读取的数据。readUint16() 读取 u2 类型数据，代码如下：

```
func (self *ClassReader) readUint16() uint16 {
   val := binary.BigEndian.Uint16(self.data)
   self.data = self.data[2:]
   return val
}
```

Go 标准库 encoding/binary 包中定义了一个变量 BigEndian，正好可以从 []byte 中解码多字节数据。readUint32() 读取 u4 类型数据，代码如下：

```
func (self *ClassReader) readUint32() uint32 {
    val := binary.BigEndian.Uint32(self.data)
    self.data = self.data[4:]
    return val
}
```

readUint64() 读取 uint64（Java 虚拟机规范并没有定义 u8）类型数据，代码如下：

```
func (self *ClassReader) readUint64() uint64 {
    val := binary.BigEndian.Uint64(self.data)
    self.data = self.data[8:]
    return val
}
```

readUint16s() 读取 uint16 表，表的大小由开头的 uint16 数据指出，代码如下：

```
func (self *ClassReader) readUint16s() []uint16 {
    n := self.readUint16()
    s := make([]uint16, n)
    for i := range s {
        s[i] = self.readUint16()
    }
    return s
}
```

最后一个方法是 readBytes()，用于读取指定数量的字节，代码如下：

```
func (self *ClassReader) readBytes(n uint32) []byte {
    bytes := self.data[:n]
    self.data = self.data[n:]
    return bytes
}
```

3.2.2　整体结构

有了 ClassReader，可以开始解析 class 文件了。在 ch03\classfile 目录下创建 class_file.go 文件，在其中定义 ClassFile 结构体，代码如下：

```
package classfile

import "fmt"

type ClassFile struct {
```

```
    //magic          uint32
    minorVersion     uint16
    majorVersion     uint16
    constantPool     ConstantPool
    accessFlags      uint16
    thisClass        uint16
    superClass       uint16
    interfaces       []uint16
    fields           []*MemberInfo
    methods          []*MemberInfo
    attributes       []AttributeInfo
}
```

ClassFile 结构体如实反映了 Java 虚拟机规范定义的 class 文件格式。还会在 class_file.go 文件中实现一系列函数和方法，列举如下：

```
func Parse(classData []byte) (cf *ClassFile, err error) {...}
func (self *ClassFile) read(reader *ClassReader) {...}
func (self *ClassFile) readAndCheckMagic(reader *ClassReader) {...}
func (self *ClassFile) readAndCheckVersion(reader *ClassReader) {...}
func (self *ClassFile) MinorVersion() uint16 {...} // getter
func (self *ClassFile) MajorVersion() uint16 {...} // getter
func (self *ClassFile) ConstantPool() ConstantPool {...} // getter
func (self *ClassFile) AccessFlags() uint16 {...} // getter
func (self *ClassFile) Fields() []*MemberInfo {...} // getter
func (self *ClassFile) Methods() []*MemberInfo {...} // getter
func (self *ClassFile) ClassName() string {...}
func (self *ClassFile) SuperClassName() string {...}
func (self *ClassFile) InterfaceNames() []string {...}
```

相比 Java 语言，Go 的访问控制非常简单：只有公开和私有两种。所有首字母大写的类型、结构体、字段、变量、函数、方法等都是公开的，可供其他包使用。首字母小写则是私有的，只能在包内部使用。在本书的代码中，尽量只公开必要的变量、字段、函数和方法等。但是为了提高代码可读性，所有的结构体都是公开的，也就是首字母是大写的。

Parse() 函数把 []byte 解析成 ClassFile 结构体，代码如下：

```
func Parse(classData []byte) (cf *ClassFile, err error) {
    defer func() {
        if r := recover(); r != nil {
            var ok bool
            err, ok = r.(error)
            if !ok {
                err = fmt.Errorf("%v", r)
            }
        }
```

```
    }()

    cr := &ClassReader{classData}
    cf = &ClassFile{}
    cf.read(cr)
    return
}
```

Go 语言没有异常处理机制，只有一个 panic-recover 机制。read() 方法依次调用其他方法解析 class 文件，代码如下：

```
func (self *ClassFile) read(reader *ClassReader) {
    self.readAndCheckMagic(reader) // 见 3.2.3
    self.readAndCheckVersion(reader) // 见 3.2.4
    self.constantPool = readConstantPool(reader) // 见 3.3
    self.accessFlags = reader.readUint16()
    self.thisClass = reader.readUint16()
    self.superClass = reader.readUint16()
    self.interfaces = reader.readUint16s()
    self.fields = readMembers(reader, self.constantPool) // 见 3.2.8
    self.methods = readMembers(reader, self.constantPool)
    self.attributes = readAttributes(reader, self.constantPool) // 见 3.4
}
```

MajorVersion() 等 6 个方法是 Getter 方法，把结构体的字段暴露给其他包使用。MajorVersion() 的代码如下：

```
func (self *ClassFile) MajorVersion() uint16 {
    return self.majorVersion
}
```

和 Java 有所不同，Go 的 Getter 方法不以“get”开头。由于 Getter 方法非常简单，只是返回字段而已，为了节约篇幅，后文中不再给出 Getter 方法的代码。ClassName() 从常量池查找类名，代码如下：

```
func (self *ClassFile) ClassName() string {
    return self.constantPool.getClassName(self.thisClass)
}
```

SuperClassName() 从常量池查找超类名，代码如下：

```
func (self *ClassFile) SuperClassName() string {
    if self.superClass > 0 {
        return self.constantPool.getClassName(self.superClass)
    }
    return "" // 只有 java.lang.Object 没有超类
}
```

InterfaceNames() 从常量池查找接口名，代码如下：

```
func (self *ClassFile) InterfaceNames() []string {
    interfaceNames := make([]string, len(self.interfaces))
    for i, cpIndex := range self.interfaces {
        interfaceNames[i] = self.constantPool.getClassName(cpIndex)
    }
    return interfaceNames
}
```

下面详细介绍 class 文件的各个部分（常量池和属性表比较复杂，放到 3.3 和 3.4 节单独讨论）。

3.2.3 魔数

很多文件格式都会规定满足该格式的文件必须以某几个固定字节开头，这几个字节主要起标识作用，叫作魔数（magic number）。例如 PDF 文件以 4 字节“%PDF”（0x25、0x50、0x44、0x46）开头，ZIP 文件以 2 字节“PK”（0x50、0x4B）开头。class 文件的魔数是“0xCAFEBABE”。readAndCheckMagic() 方法的代码如下：

```
func (self *ClassFile) readAndCheckMagic(reader *ClassReader) {
    magic := reader.readUint32()
    if magic != 0xCAFEBABE {
        panic("java.lang.ClassFormatError: magic!")
    }
}
```

Java 虚拟机规范规定，如果加载的 class 文件不符合要求的格式，Java 虚拟机实现就抛出 java.lang.ClassFormatError 异常。但是因为我们才刚刚开始编写虚拟机，还无法抛出异常，所以暂时先调用 panic() 方法终止程序执行。用 classpy 打开 ClassFileTest.class 文件，可以看到，开头 4 字节确实是 0xCAFEBABE，如图 3-1 所示。

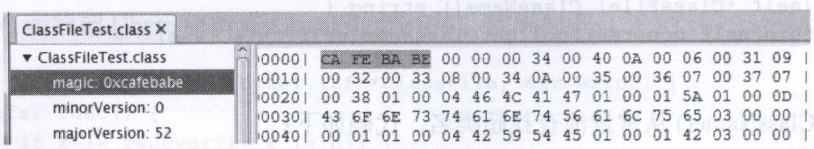


图 3-1 用 classpy 观察魔数

3.2.4 版本号

魔数之后是class文件的次版本号和主版本号，都是u2类型。假设某class文件的主版本号是M，次版本号是m，那么完整的版本号可以表示成“M.m”的形式。次版本号只在J2SE 1.2之前用过，从1.2开始基本上就没什么用了（都是0）。主版本号在J2SE 1.2之前是45，从1.2开始，每次有大的Java版本发布，都会加1。表3-2列出了到本书写作为止，使用过的class文件版本号。

表3-2 class文件版本号

Java版本	class文件版本号	Java版本	class文件版本号
JDK 1.0.2	45.0 ~ 45.3	Java SE 5.0	49.0
JDK 1.1	45.0 ~ 45.65535	Java SE 6	50.0
J2SE 1.2	46.0	Java SE 7	51.0
J2SE 1.3	47.0	Java SE 8	52.0
J2SE 1.4	48.0		

特定的Java虚拟机实现只能支持版本号在某个范围内的class文件。Oracle的实现是完全向后兼容的，比如Java SE 8支持版本号为45.0 ~ 52.0的class文件。如果版本号不在支持的范围内，Java虚拟机实现就抛出java.lang.UnsupportedClassVersionError异常。我们参考Java 8，支持版本号为45.0 ~ 52.0的class文件。如果遇到其他版本号，暂时先调用panic()方法终止程序执行。下面是readAndCheckVersion()方法的代码。

```
func (self *ClassFile) readAndCheckVersion(reader *ClassReader) {
    self.minorVersion = reader.readUint16()
    self.majorVersion = reader.readUint16()
    switch self.majorVersion {
    case 45:
        return
    case 46, 47, 48, 49, 50, 51, 52:
        if self.minorVersion == 0 {
            return
        }
    }
    panic("java.lang.UnsupportedClassVersionError!")
}
```

因为笔者使用JDK8编译ClassFileTest类，所以主版本号是52（0x34），次版本号是0，如图3-2所示。

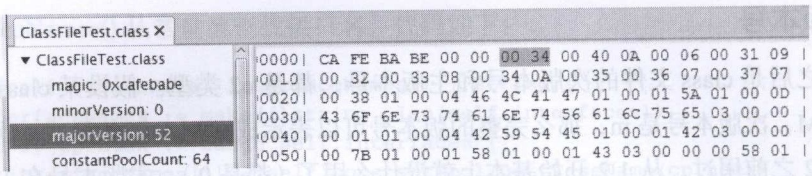


图 3-2 用 classpy 观察版本号

3.2.5 类访问标志

版本号之后是常量池，但是由于常量池比较复杂，所以放到 3.3 节介绍。常量池之后是类访问标志，这是一个 16 位的“bitmask”，指出 class 文件定义的是类还是接口，访问级别是 public 还是 private，等等。本章只对 class 文件进行初步解析，并不做完整验证，所以只是读取类访问标志以备后用。第 6 章会详细讨论访问标志。ClassFileTest 的类访问标志是 0x21，如图 3-3 所示。

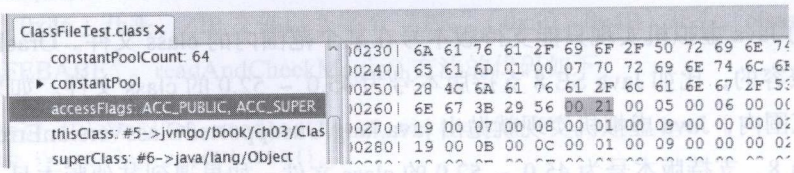


图 3-3 用 classpy 观察类访问标志

3.2.6 类和超类索引

类访问标志之后是两个 u2 类型的常量池索引，分别给出类名和超类名。class 文件存储的类名类似完全限定名，但是把点换成了斜线，Java 语言规范把这种名字叫作二进制名（binary names）。因为每个类都有名字，所以 thisClass 必须是有效的常量池索引。除 java.lang.Object 之外，其他类都有超类，所以 superClass 只在 Object.class 中是 0，在其他 class 文件中必须是有效的常量池索引。如图 3-4 所示，ClassFileTest 的类索引是 5，超类索引是 6。

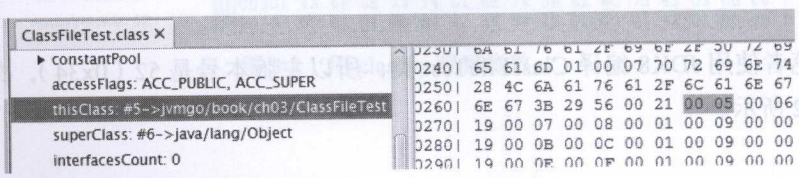


图 3-4 用 classpy 观察类和超类索引

3.2.7　接口索引表

类和超类索引后面是接口索引表，表中存放的也是常量池索引，给出该类实现的所有接口的名字。ClassFileTest 没有实现接口，所以接口表是空的，如图 3-5 所示。

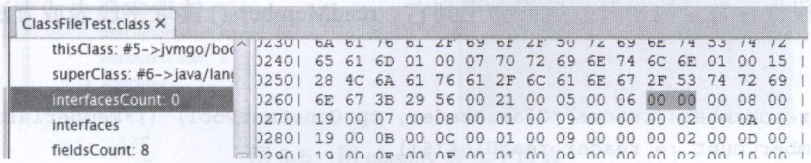


图 3-5　用 classpy 观察接口索引表

3.2.8　字段和方法表

接口索引表之后是字段表和方法表，分别存储字段和方法信息。字段和方法的基本结构大致相同，差别仅在于属性表。下面是 Java 虚拟机规范给出的字段结构定义。

```
field_info {
    u2                  access_flags;
    u2                  name_index;
    u2                  descriptor_index;
    u2                  attributes_count;
    attribute_info      attributes[attributes_count];
}
```

和类一样，字段和方法也有自己的访问标志。访问标志之后是一个常量池索引，给出字段名或方法名，然后又是一个常量池索引，给出字段或方法的描述符，最后是属性表。为了避免重复代码，用一个结构体统一表示字段和方法。在 ch03\classfile 目录中创建 member_info.go 文件，在其中定义 MemberInfo 结构体，代码如下：

```
package classfile

type MemberInfo struct {
    cp              ConstantPool
    accessFlags     uint16
    nameIndex       uint16
    descriptorIndex uint16
    attributes      []AttributeInfo
}

func readMembers(reader *ClassReader, cp ConstantPool) []*MemberInfo {...}
```

```
func readMember(reader *ClassReader, cp ConstantPool) *MemberInfo {...}
func (self *MemberInfo) AccessFlags() uint16 {...} // getter
func (self *MemberInfo) Name() string {...}
func (self *MemberInfo) Descriptor() string {...}
```

cp 字段保存常量池指针，后面会用到它。readMembers() 读取字段表或方法表，代码如下：

```
func readMembers(reader *ClassReader, cp ConstantPool) []*MemberInfo {
    memberCount := reader.readUint16()
    members := make([]*MemberInfo, memberCount)
    for i := range members {
        members[i] = readMember(reader, cp)
    }
    return members
}
```

readMember() 函数读取字段或方法数据，代码如下：

```
func readMember(reader *ClassReader, cp ConstantPool) *MemberInfo {
    return &MemberInfo{
        cp:              cp,
        accessFlags:     reader.readUint16(),
        nameIndex:       reader.readUint16(),
        descriptorIndex: reader.readUint16(),
        attributes:      readAttributes(reader, cp), // 见 3.4
    }
}
```

属性表和 readAttributes() 函数将在 3.4 节介绍。Name() 从常量池查找字段或方法名，Descriptor() 从常量池查找字段或方法描述符，代码如下：

```
func (self *MemberInfo) Name() string {
    return self.cp.getUtf8(self.nameIndex)
}
func (self *MemberInfo) Descriptor() string {
    return self.cp.getUtf8(self.descriptorIndex)
}
```

第 6 章会进一步讨论字段和方法。ClassFileTest 有 8 个字段和两个方法（其中 <init> 是编译器生成的默认构造函数），如图 3-6 和图 3-7 所示。

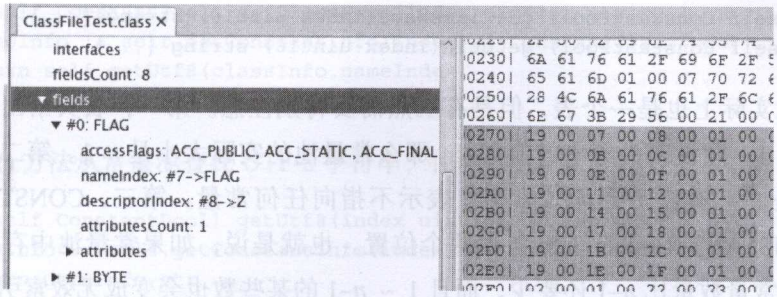


图 3-6 用 classpy 观察字段表

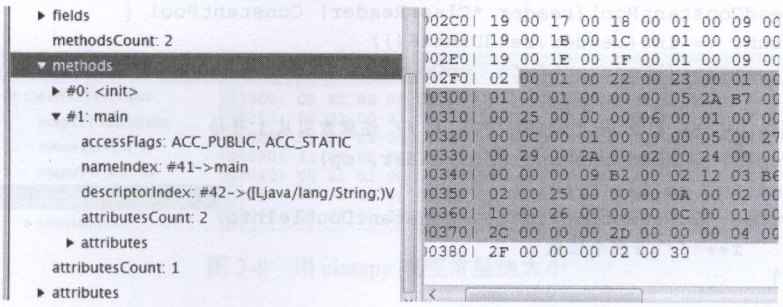


图 3-7 用 classpy 观察方法表

3.3 解析常量池

常量池占据了 class 文件很大一部分数据，里面存放着各式各样的常量信息，包括数字和字符串常量、类和接口名、字段和方法名，等等。本节将详细介绍常量池和各种常量。

3.3.1 ConstantPool 结构体

在 ch03\classfile 目录下创建 constant_pool.go 文件，在里面定义 ConstantPool 类型，代码如下所示：

```
package classfile

type ConstantPool []ConstantInfo

func readConstantPool(reader *ClassReader) ConstantPool {...}
func (self ConstantPool) getConstantInfo(index uint16) ConstantInfo {...}
func (self ConstantPool) getNameAndType(index uint16) (string, string) {...}
```

```
func (self ConstantPool) getClassName(index uint16) string {...}
func (self ConstantPool) getUtf8(index uint16) string {...}
```

常量池实际上也是一个表，但是有三点需要特别注意。第一，表头给出的常量池大小比实际大 1。假设表头给出的值是 n，那么常量池的实际大小是 n–1。第二，有效的常量池索引是 1 ~ n–1。0 是无效索引，表示不指向任何常量。第三，CONSTANT_Long_info 和 CONSTANT_Double_info 各占两个位置。也就是说，如果常量池中存在这两种常量，实际的常量数量比 n–1 还要少，而且 1 ~ n–1 的某些数也会变成无效索引。常量池由 readConstantPool() 函数读取，代码如下：

```
func readConstantPool(reader *ClassReader) ConstantPool {
    cpCount := int(reader.readUint16())
    cp := make([]ConstantInfo, cpCount)

    for i := 1; i < cpCount; i++ { // 注意索引从 1 开始
        cp[i] = readConstantInfo(reader, cp)
        switch cp[i].(type) {
        case *ConstantLongInfo, *ConstantDoubleInfo:
            i++ // 占两个位置
        }
    }

    return cp
}
```

getConstantInfo() 方法按索引查找常量，代码如下：

```
func (self ConstantPool) getConstantInfo(index uint16) ConstantInfo {
    if cpInfo := self[index]; cpInfo != nil {
        return cpInfo
    }
    panic("Invalid constant pool index!")
}
```

getNameAndType() 方法从常量池查找字段或方法的名字和描述符，代码如下：

```
func (self ConstantPool) getNameAndType(index uint16) (string, string) {
    ntInfo := self.getConstantInfo(index).(*ConstantNameAndTypeInfo)
    name := self.getUtf8(ntInfo.nameIndex)
    _type := self.getUtf8(ntInfo.descriptorIndex)
    return name, _type
}
```

getClassName() 方法从常量池查找类名，代码如下：

```
func (self ConstantPool) getClassName(index uint16) string {
    classInfo := self.getConstantInfo(index).(*ConstantClassInfo)
    return self.getUtf8(classInfo.nameIndex)
}
```

getUtf8()方法从常量池查找UTF-8字符串，代码如下：

```
func (self ConstantPool) getUtf8(index uint16) string {
    utf8Info := self.getConstantInfo(index).(*ConstantUtf8Info)
    return utf8Info.str
}
```

ClassFileTest的常量池大小是61如图3-8所示。

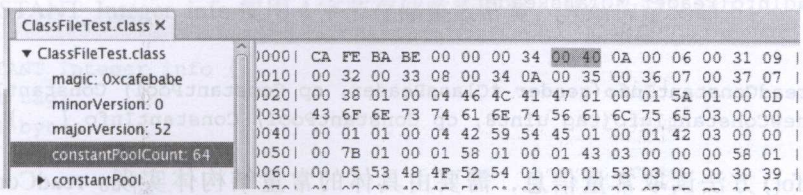


图3-8 用classpy观察常量池大小

3.3.2 ConstantInfo接口

由于常量池中存放的信息各不相同，所以每种常量的格式也不同。常量数据的第一字节是tag，用来区分常量类型。下面是Java虚拟机规范给出的常量结构。

```
cp_info {
    u1 tag;
    u1 info[];
}
```

Java虚拟机规范一共定义了14种常量。在ch03\classfile目录下创建constant_info.go文件，在其中定义tag常量值，代码如下：

```
package classfile

// tag常量值定义
const (
    CONSTANT_Class              = 7
    CONSTANT_Fieldref           = 9
    CONSTANT_Methodref          = 10
    CONSTANT_InterfaceMethodref = 11
    CONSTANT_String             = 8
```

```
    CONSTANT_Integer            = 3
    CONSTANT_Float              = 4
    CONSTANT_Long               = 5
    CONSTANT_Double             = 6
    CONSTANT_NameAndType        = 12
    CONSTANT_Utf8               = 1
    CONSTANT_MethodHandle       = 15
    CONSTANT_MethodType         = 16
    CONSTANT_InvokeDynamic      = 18
)
```

继续编辑 constant_pool.go，定义 ConstantInfo 接口来表示常量信息，代码如下：

```
type ConstantInfo interface {
    readInfo(reader *ClassReader)
}

func readConstantInfo(reader *ClassReader, cp ConstantPool) ConstantInfo {...}
func newConstantInfo(tag uint8, cp ConstantPool) ConstantInfo {...}
```

readInfo() 方法读取常量信息，需要由具体的常量结构体实现。readConstantInfo() 函数先读出 tag 值，然后调用 newConstantInfo() 函数创建具体的常量，最后调用常量的 readInfo() 方法读取常量信息，代码如下：

```
func readConstantInfo(reader *ClassReader, cp ConstantPool) ConstantInfo {
    tag := reader.readUint8()
    c := newConstantInfo(tag, cp)
    c.readInfo(reader)
    return c
}
```

newConstantInfo() 根据 tag 值创建具体的常量，代码如下：

```
func newConstantInfo(tag uint8, cp ConstantPool) ConstantInfo {
    switch tag {
    case CONSTANT_Integer: return &ConstantIntegerInfo{}
    case CONSTANT_Float: return &ConstantFloatInfo{}
    case CONSTANT_Long: return &ConstantLongInfo{}
    case CONSTANT_Double: return &ConstantDoubleInfo{}
    case CONSTANT_Utf8: return &ConstantUtf8Info{}
    case CONSTANT_String: return &ConstantStringInfo{cp: cp}
    case CONSTANT_Class: return &ConstantClassInfo{cp: cp}
    case CONSTANT_Fieldref:
        return &ConstantFieldrefInfo{ConstantMemberrefInfo{cp: cp}}
    case CONSTANT_Methodref:
        return &ConstantMethodrefInfo{ConstantMemberrefInfo{cp: cp}}
    case CONSTANT_InterfaceMethodref:
```

```
        return &ConstantInterfaceMethodrefInfo{ConstantMemberrefInfo{cp: cp}}
    case CONSTANT_NameAndType: return &ConstantNameAndTypeInfo{}
    case CONSTANT_MethodType: return &ConstantMethodTypeInfo{}
    case CONSTANT_MethodHandle: return &ConstantMethodHandleInfo{}
    case CONSTANT_InvokeDynamic: return &ConstantInvokeDynamicInfo{}
    default: panic("java.lang.ClassFormatError: constant pool tag!")
    }
}
```

下面的小节详细介绍各种常量。

3.3.3 CONSTANT_Integer_info

CONSTANT_Integer_info使用4字节存储整数常量，其结构定义如下：

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}
```

CONSTANT_Integer_info和后面将要介绍的其他三种数字常量无论是结构，还是实现，都非常相似，所以把它们定义在同一个文件中。在ch03\classfile目录下创建cp_numeric.go文件，在其中定义ConstantIntegerInfo结构体，代码如下：

```
package classfile

import "math"

type ConstantIntegerInfo struct {
    val int32
}

func (self *ConstantIntegerInfo) readInfo(reader *ClassReader) {...}
```

readInfo()先读取一个uint32数据，然后把它转型成int32类型，代码如下：

```
func (self *ConstantIntegerInfo) readInfo(reader *ClassReader) {
    bytes := reader.readUint32()
    self.val = int32(bytes)
}
```

CONSTANT_Integer_info正好可以容纳一个Java的int型常量，但实际上比int更小的boolean、byte、short和char类型的常量也放在CONSTANT_Integer_info中。编译器给ClassFileTest类的INT字段生成了一个CONSTANT_Integer_info常量，如图3-9所示。

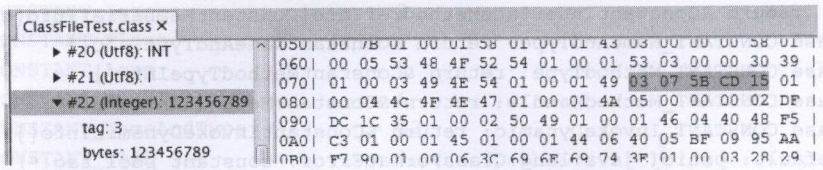


图 3-9 用 classpy 观察 CONSTANT_Integer_info 常量

3.3.4 CONSTANT_Float_info

CONSTANT_Float_info 使用 4 字节存储 IEEE754 单精度浮点数常量，结构如下：

```
CONSTANT_Float_info {
    u1 tag;
    u4 bytes;
}
```

在 cp_numeric.go 文件中定义 ConstantFloatInfo 结构体，代码如下：

```
type ConstantFloatInfo struct {
    val float32
}
func (self *ConstantFloatInfo) readInfo(reader *ClassReader) {
    bytes := reader.readUint32()
    self.val = math.Float32frombits(bytes)
}
```

readInfo() 先读取一个 uint32 数据，然后调用 math 包的 Float32frombits() 函数把它转换成 float32 类型。编译器给 ClassFileTest 类的 PI 字段生成了一个 CONSTANT_Float_info 常量，如图 3-10 所示。

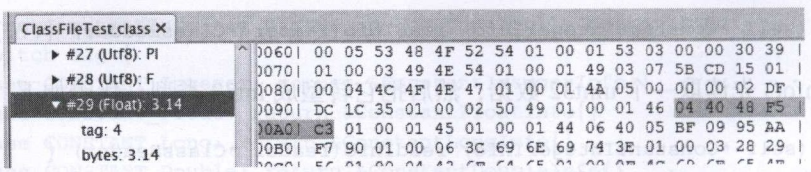


图 3-10 用 classpy 观察 CONSTANT_Float_info 常量

3.3.5 CONSTANT_Long_info

CONSTANT_Long_info 使用 8 字节存储整数常量，结构如下：

```
CONSTANT_Long_info {
```

```
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

在 cp_numeric.go 文件中定义 ConstantLongInfo 结构体，代码如下：

```
type ConstantLongInfo struct {
    val int64
}
func (self *ConstantLongInfo) readInfo(reader *ClassReader) {
    bytes := reader.readUint64()
    self.val = int64(bytes)
}
```

readInfo() 先读取一个 uint64 数据，然后把它转型成 int64 类型。编译器给 ClassFileTest 类的 LONG 字段生成了一个 CONSTANT_Long_info 常量，如图 3-11 所示。

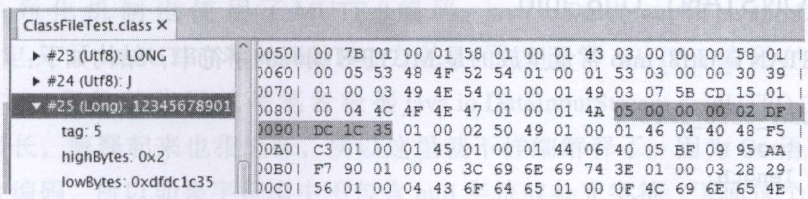


图 3-11　用 classpy 观察 CONSTANT_Long_info 常量

3.3.6　CONSTANT_Double_info

最后一个数字常量是 CONSTANT_Double_info，使用 8 字节存储 IEEE754 双精度浮点数，结构如下：

```
CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

在 cp_numeric.go 文件中定义 ConstantDoubleInfo 结构体，代码如下：

```
type ConstantDoubleInfo struct {
    val float64
}
func (self *ConstantDoubleInfo) readInfo(reader *ClassReader) {
    bytes := reader.readUint64()
```

```
    self.val = math.Float64frombits(bytes)
}
```

readInfo() 先读取一个 uint64 数据，然后调用 math 包的 Float64frombits() 函数把它转换成 float64 类型。编译器给 ClassFileTest 类的 E 字段生成了一个 CONSTANT_Double_info 常量，如图 3-12 所示。

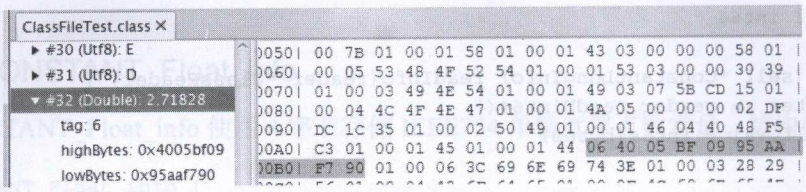


图 3-12 用 classpy 观察 CONSTANT_Double_info 常量

3.3.7 CONSTANT_Utf8_info

CONSTANT_Utf8_info 常量里放的是 MUTF-8 编码的字符串，结构如下：

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

注意，字符串在 class 文件中是以 MUTF-8（Modified UTF-8）方式编码的。但为什么没有用标准的 UTF-8 编码方式，笔者没有找到明确的原因[⊖]。MUTF-8 编码方式和 UTF-8 大致相同，但并不兼容。差别有两点：一是 null 字符（代码点 U+0000）会被编码成 2 字节：0xC0、0x80；二是补充字符（Supplementary Characters，代码点大于 U+FFFF 的 Unicode 字符）是按 UTF-16 拆分为代理对（Surrogate Pair）分别编码的。具体细节超出了本章的讨论范围，有兴趣的读者可以阅读 Java 虚拟机规范和 Unicode 规范的相关章节[⊜]。

在 ch03\classfile 目录下创建 cp_utf8.go 文件，在其中定义 ConstantUtf8Info 结构体，代码如下：

```
package classfile
```

⊖ 这个链接中有一些线索：http://stackoverflow.com/questions/15440584/why-does-java-use-modified-utf-8-instead-of-utf-8。

⊜ 或者这篇文章：http://www.oracle.com/technetwork/articles/javase/supplementary-142654.html。

```
import "fmt"
import "unicode/utf16"

type ConstantUtf8Info struct {
    str string
}

func (self *ConstantUtf8Info) readInfo(reader *ClassReader) {}
```

readInfo()方法先读取出[]byte，然后调用decodeMUTF8()函数把它解码成Go字符串，代码如下：

```
func (self *ConstantUtf8Info) readInfo(reader *ClassReader) {
   length := uint32(reader.readUint16())
   bytes := reader.readBytes(length)
   self.str = decodeMUTF8(bytes)
}
```

Java序列化机制也使用了MUTF-8编码。java.io.DataInput和java.io.DataOutput接口分别定义了readUTF()和writeUTF()方法，可以读写MUTF-8编码的字符串。decodeMUTF8()函数的代码就是笔者根据java.io.DataInputStream.readUTF()方法改写的。代码很长，解释起来也很乏味，所以这里就不详细解释了。因为Go语言字符串使用UTF-8编码，所以如果字符串中不包含null字符或补充字符，下面这个简化版的readMUTF8()也是可以工作的。

```
// 简化版，完整版请阅读本章源代码
func decodeMUTF8(bytes []byte) string {
   return string(bytes)
}
```

相信细心的读者在前面的截图中已经看到了，字段名、字段描述符等就是以字符串的形式存储在class文件中的，如字段PI对应的CONSTANT_Utf8_info常量，如图3-13所示。

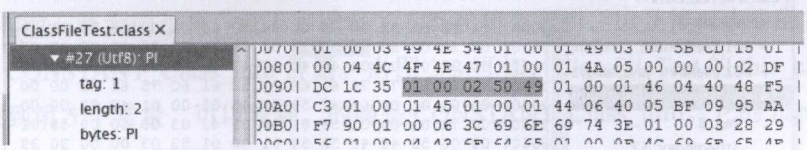


图3-13 用classpy观察CONSTANT_Utf8_info常量

3.3.8 CONSTANT_String_info

CONSTANT_String_info常量表示java.lang.String字面量，结构如下：

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

可以看到，CONSTANT_String_info 本身并不存放字符串数据，只存了常量池索引，这个索引指向一个 CONSTANT_Utf8_info 常量。在 ch03\classfile 目录下创建 cp_string.go 文件，在其中定义 ConstantStringInfo 结构体，代码如下：

```
package classfile

type ConstantStringInfo struct {
    cp              ConstantPool
    stringIndex     uint16
}

func (self *ConstantStringInfo) readInfo(reader *ClassReader) {...}
func (self *ConstantStringInfo) String() string {...}
```

readInfo() 方法读取常量池索引，代码如下：

```
func (self *ConstantStringInfo) readInfo(reader *ClassReader) {
    self.stringIndex = reader.readUint16()
}
```

String() 方法按索引从常量池中查找字符串，代码如下：

```
func (self *ConstantStringInfo) String() string {
    return self.cp.getUtf8(self.stringIndex)
}
```

ClassFileTest 的 main() 方法使用了字符串字面量“Hello,World!”，对应的 CONSTANT_String_info 常量如图 3-14 所示。

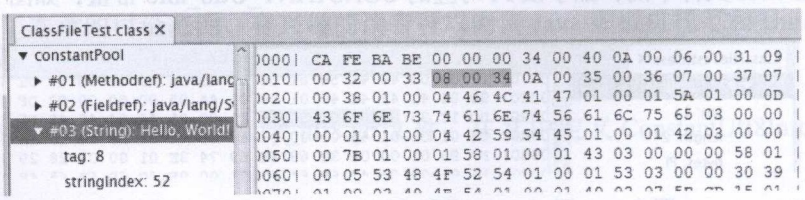


图 3-14　用 classpy 观察 CONSTANT_String_info 常量

可以看到，string_index 是 52（0x34）。我们按图索骥，从常量池中找出第 52 个常量，确实是个 CONSTANT_Utf8_info，如图 3-15 所示。

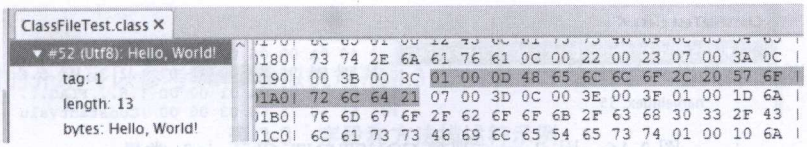


图 3-15　用 classpy 观察 CONSTANT_String_info 常量（2）

3.3.9　CONSTANT_Class_info

CONSTANT_Class_info 常量表示类或者接口的符号引用，结构如下：

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

和 CONSTANT_String_info 类似，name_index 是常量池索引，指向 CONSTANT_Utf8_info 常量。在 ch03\classfile 目录下创建 cp_class.go 文件，在其中定义 ConstantClassInfo 结构体，代码如下：

```
package classfile

type ConstantClassInfo struct {
    cp              ConstantPool
    nameIndex       uint16
}

func (self *ConstantClassInfo) readInfo(reader *ClassReader) {
    self.nameIndex = reader.readUint16()
}
func (self *ConstantClassInfo) Name() string {
    return self.cp.getUtf8(self.nameIndex)
}
```

代码和前一节大同小异，就不多解释了。类和超类索引，以及接口表中的接口索引指向的都是 CONSTANT_Class_info 常量。由图 3-3 可知，ClassFileTest 的 this_class 索引是 5。我们找到第 5 个常量，可以看到，的确是 CONSTANT_Class_info。它的 name_index 是 55（0x37），如图 3-16 所示。

再看第 55 个常量，也的确是 CONSTANT_Utf_info，如图 3-17 所示。

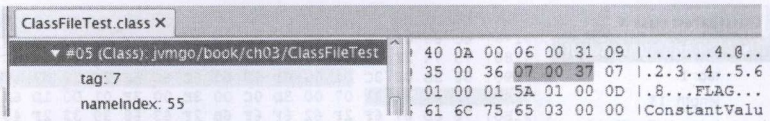


图 3-16 用 classpy 观察 CONSTANT_Class_info 常量

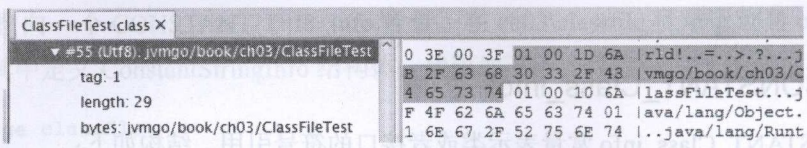


图 3-17 用 classpy 观察 CONSTANT_Class_info 常量（2）

3.3.10 CONSTANT_NameAndType_info

CONSTANT_NameAndType_info 给出字段或方法的名称和描述符。CONSTANT_Class_info 和 CONSTANT_NameAndType_info 加在一起可以唯一确定一个字段或者方法。其结构如下：

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

字段或方法名由 name_index 给出，字段或方法的描述符由 descriptor_index 给出。name_index 和 descriptor_index 都是常量池索引，指向 CONSTANT_Utf8_info 常量。字段和方法名就是代码中出现的（或者编译器生成的）字段或方法的名字。Java 虚拟机规范定义了一种简单的语法来描述字段和方法，可以根据下面的规则生成描述符。

1）类型描述符。

①基本类型 byte、short、char、int、long、float 和 double 的描述符是单个字母，分别对应 B、S、C、I、J、F 和 D。注意，long 的描述符是 J 而不是 L。

②引用类型的描述符是 L +类的完全限定名+分号。

③数组类型的描述符是 [+数组元素类型描述符。

2）字段描述符就是字段类型的描述符。

3）方法描述符是（分号分隔的参数类型描述符）+ 返回值类型描述符，其中 void 返

回值由单个字母 V 表示。

更详细的介绍可以参考 Java 虚拟机规范 4.3 节。表 3-3 给出了一些具体的例子。

表 3-3　字段和方法描述符示例

字段描述符	字段类型	方法描述符	方　法
S	short	()V	void run()
Ljava.lang.Object;	java.lang.Object	()Ljava.lang.String;	String toString()
[I	int[]	(Ljava.lang.String;)V	void main(String[] args)
[[D	double[][]	(FF)F	int max(float x, float y)
[Ljava.lang.String;	java.lang.Object[]	([JJ)I	int binarySearch(long[] a, long key)

我们都知道，Java 语言支持方法重载（override），不同的方法可以有相同的名字，只要参数列表不同即可。这就是为什么 CONSTANT_NameAndType_info 结构要同时包含名称和描述符的原因。那么字段呢？Java 是不能定义多个同名字段的，哪怕它们的类型各不相同。这只是 Java 语法的限制而已，从 class 文件的层面来看，是完全可以支持这点的。

在 ch03\classfile 目录下创建 cp_name_and_type.go 文件，在其中定义 ConstantNameAndTypeInfo 结构体，代码如下：

```
package classfile

type ConstantNameAndTypeInfo struct {
    nameIndex       uint16
    descriptorIndex uint16
}

func (self *ConstantNameAndTypeInfo) readInfo(reader *ClassReader) {
    self.nameIndex = reader.readUint16()
    self.descriptorIndex = reader.readUint16()
}
```

代码比较简单，就不多解释了。

3.3.11　CONSTANT_Fieldref_info、CONSTANT_Methodref_info 和 CONSTANT_InterfaceMethodref_info

CONSTANT_Fieldref_info 表示字段符号引用，CONSTANT_Methodref_info 表示普通（非接口）方法符号引用，CONSTANT_InterfaceMethodref_info 表示接口方法符号引用。这三种常量结构一模一样，为了节约篇幅，下面只给出 CONSTANT_Fieldref_info 的结构。

```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

class_index 和 name_and_type_index 都是常量池索引，分别指向 CONSTANT_Class_info 和 CONSTANT_NameAndType_info 常量。先定义一个统一的结构体 ConstantMemberrefInfo 来表示这 3 种常量。在 ch03\classfile 目录下创建 cp_member_ref.go 文件，把下面的代码输入进去。

```
package classfile

type ConstantMemberrefInfo struct {
    cp                  ConstantPool
    classIndex          uint16
    nameAndTypeIndex    uint16
}

func (self *ConstantMemberrefInfo) readInfo(reader *ClassReader) {
    self.classIndex = reader.readUint16()
    self.nameAndTypeIndex = reader.readUint16()
}

func (self *ConstantMemberrefInfo) ClassName() string {
    return self.cp.getClassName(self.classIndex)
}
func (self *ConstantMemberrefInfo) NameAndDescriptor() (string, string) {
    return self.cp.getNameAndType(self.nameAndTypeIndex)
}
```

然后定义三个结构体“继承”ConstantMemberrefInfo。Go 语言并没有“继承”这个概念，但是可以通过结构体嵌套来模拟，代码如下：

```
type ConstantFieldrefInfo struct{ ConstantMemberrefInfo }
type ConstantMethodrefInfo struct{ ConstantMemberrefInfo }
type ConstantInterfaceMethodrefInfo struct{ ConstantMemberrefInfo }
```

ClassFileTest 类的 main() 方法使用了 java.lang.System 类的 out 字段，该字段由常量池第 2 项指出，如图 3-18 所示。

可以看到，class_index 是 50（0x32），name_and_type_index 是 51（0x33）。我们找到第 50 和第 51 个常量，可以看到，确实是 CONSTANT_Class_info 和 CONSTANT_NameAndType_info，如图 3-19 所示。

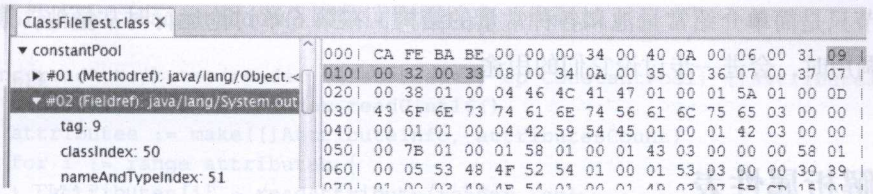


图 3-18　用 classpy 观察 CONSTANT_Fieldref_info 常量

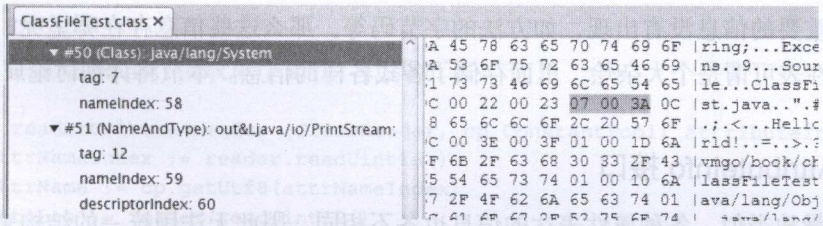


图 3-19　用 classpy 观察 CONSTANT_Fieldref_info 常量（2）

3.3.12　常量池小结

还有三个常量没有介绍：CONSTANT_MethodType_info、CONSTANT_MethodHandle_info 和 CONSTANT_InvokeDynamic_info。它们是 Java SE 7 才添加到 class 文件中的，目的是支持新增的 invokedynamic 指令。本书不讨论 invokedynamic 指令，所以解析这三个常量的代码就不在这里介绍了。代码也非常简单，有兴趣的读者可以阅读随书源代码中的 ch03\cp_invoke_dynamic.go 文件。

可以把常量池中的常量分为两类：字面量（literal）和符号引用（symbolic reference）。字面量包括数字常量和字符串常量，符号引用包括类和接口名、字段和方法信息等。除了字面量，其他常量都是通过索引直接或间接指向 CONSTANT_Utf8_info 常量，以 CONSTANT_Fieldref_info 为例，如图 3-20 所示。

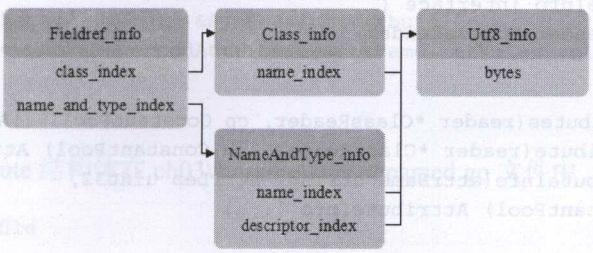


图 3-20　常量引用关系

本节只是简单介绍常量池和各种常量的结构，在第 6 章讨论运行时常量池，第 7 章讨论方法调用时，会进一步讨论它们的用途。

3.4 解析属性表

3.2 节大致勾勒出了 class 文件的结构，3.3 节介绍了常量池。细心的读者一定会发现，还有一些重要的信息没有出现，如方法的字节码等。那么这些信息存在哪里呢？答案是属性表。属性表可谓是个大杂烩，里面存储了各式各样的信息。本节将详细讨论属性表。

3.4.1 AttributeInfo 接口

和常量池类似，各种属性表达的信息也各不相同，因此无法用统一的结构来定义。不同之处在于，常量是由 Java 虚拟机规范严格定义的，共有 14 种。但属性是可以扩展的，不同的虚拟机实现可以定义自己的属性类型。由于这个原因，Java 虚拟机规范没有使用 tag，而是使用属性名来区别不同的属性。属性数据放在属性名之后的 u1 表中，这样 Java 虚拟机实现就可以跳过自己无法识别的属性。属性的结构定义如下：

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

注意，属性表中存放的属性名实际上并不是编码后的字符串，而是常量池索引，指向常量池中的 CONSTANT_Utf8_info 常量。在 ch03\classfile 目录下创建 attribute_info.go 文件，在其中定义 AttributeInfo 接口，代码如下：

```
package classfile

type AttributeInfo interface {
    readInfo(reader *ClassReader)
}

func readAttributes(reader *ClassReader, cp ConstantPool) []AttributeInfo {...}
func readAttribute(reader *ClassReader, cp ConstantPool) AttributeInfo {...}
func newAttributeInfo(attrName string, attrLen uint32,
        cp ConstantPool) AttributeInfo {...}
```

和 ConstantInfo 接口一样，AttributeInfo 接口也只定义了一个 readInfo() 方法，需要由

具体的属性实现。readAttributes() 函数读取属性表，代码如下：

```
func readAttributes(reader *ClassReader, cp ConstantPool) []AttributeInfo {
    attributesCount := reader.readUint16()
    attributes := make([]AttributeInfo, attributesCount)
    for i := range attributes {
        attributes[i] = readAttribute(reader, cp)
    }
    return attributes
}
```

函数 readAttribute() 读取单个属性，代码如下：

```
func readAttribute(reader *ClassReader, cp ConstantPool) AttributeInfo {
    attrNameIndex := reader.readUint16()
    attrName := cp.getUtf8(attrNameIndex)
    attrLen := reader.readUint32()
    attrInfo := newAttributeInfo(attrName, attrLen, cp)
    attrInfo.readInfo(reader)
    return attrInfo
}
```

readAttribute() 先读取属性名索引，根据它从常量池中找到属性名，然后读取属性长度，接着调用 newAttributeInfo() 函数创建具体的属性实例。Java 虚拟机规范预定义了 23 种属性，先解析其中的 8 种。newAttributeInfo() 函数的代码如下：

```
func newAttributeInfo(attrName string, attrLen uint32,
    cp ConstantPool) AttributeInfo {

    switch attrName {
    case "Code": return &CodeAttribute{cp: cp}
    case "ConstantValue": return &ConstantValueAttribute{}
    case "Deprecated": return &DeprecatedAttribute{}
    case "Exceptions": return &ExceptionsAttribute{}
    case "LineNumberTable": return &LineNumberTableAttribute{}
    case "LocalVariableTable": return &LocalVariableTableAttribute{}
    case "SourceFile": return &SourceFileAttribute{cp: cp}
    case "Synthetic": return &SyntheticAttribute{}
    default: return &UnparsedAttribute{attrName, attrLen, nil}
    }
}
```

UnparsedAttribute 结构体在 ch03\classfile\attr_unparsed.go 文件中，代码如下：

```
package classfile

type UnparsedAttribute struct {
```

```
    name   string
    length uint32
    info   []byte
}

func (self *UnparsedAttribute) readInfo(reader *ClassReader) {
    self.info = reader.readBytes(self.length)
}
```

按照用途，23 种预定义属性可以分为三组。第一组属性是实现 Java 虚拟机所必需的，共有 5 种；第二组属性是 Java 类库所必需的，共有 12 种；第三组属性主要提供给工具使用，共有 6 种。第三组属性是可选的，也就是说可以不出现在 class 文件中。如果 class 文件中存在第三组属性，Java 虚拟机实现或者 Java 类库也是可以利用它们的，比如使用 LineNumberTable 属性在异常堆栈中显示行号。

从 class 文件演进的角度来讲，JDK1.0 时只有 6 种预定义属性，JDK1.1 增加了 3 种。J2SE 5.0 增加了 9 种属性，主要用于支持泛型和注解。Java SE 6 增加了 StackMapTable 属性，用于优化字节码验证。Java SE 7 增加了 BootstrapMethods 属性，用于支持新增的 invokedynamic 指令。Java SE 8 又增加了三种属性。表 3-5 给出了这 23 种属性出现的 Java 版本、分组以及它们在 class 文件中的位置。

表 3-4 预定义属性

属性名	Java SE	分组	位置
ConstantValue	1.0.2	1	field_info
Code	1.0.2	1	method_info
Exceptions	1.0.2	1	method_info
SourceFile	1.0.2	3	ClassFile
LineNumberTable	1.0.2	3	Code
LocalVariableTable	1.0.2	3	Code
InnerClasses	1.1	2	ClassFile
Synthetic	1.1	2	ClassFile, field_info, method_info
Deprecated	1.1	3	ClassFile, field_info, method_info
EnclosingMethod	5.0	2	ClassFile
Signature	5.0	2	ClassFile, field_info, method_info
SourceDebugExtension	5.0	3	ClassFile
LocalVariableTypeTable	5.0	3	Code
RuntimeVisibleAnnotations	5.0	2	ClassFile, field_info, method_info
RuntimeInvisibleAnnotations	5.0	2	ClassFile, field_info, method_info

（续）

属性名	Java SE	分组	位置
RuntimeVisibleParameterAnnotations	5.0	2	method_info
RuntimeInvisibleParameterAnnotations	5.0	2	method_info
AnnotationDefault	5.0	2	method_info
StackMapTable	6	1	Code
BootstrapMethods	7	1	ClassFile
RuntimeVisibleTypeAnnotations	8	2	ClassFile, field_info, method_info, Code
RuntimeInvisibleTypeAnnotations	8	2	ClassFile, field_info, method_info, Code
MethodParameters	8	2	method_info

由于篇幅的限制，下面只介绍其中的 8 种属性。

3.4.2　Deprecated 和 Synthetic 属性

Deprecated 和 Synthetic 是最简单的两种属性，仅起标记作用，不包含任何数据。这两种属性都是 JDK1.1 引入的，可以出现在 ClassFile、field_info 和 method_info 结构中，它们的结构定义如下：

```
Deprecated_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
Synthetic_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

由于不包含任何数据，所以 attribute_length 的值必须是 0。Deprecated 属性用于指出类、接口、字段或方法已经不建议使用，编译器等工具可以根据 Deprecated 属性输出警告信息。J2SE 5.0 之前可以使用 Javadoc 提供的 @deprecated 标签指示编译器给类、接口、字段或方法添加 Deprecated 属性，语法格式如下：

```
/** @deprecated */
public void oldMethod() {...}
```

从 J2SE 5.0 开始，也可以使用 @Deprecated 注解，语法格式如下：

```
@Deprecated
public void oldMethod() {}
```

Synthetic 属性用来标记源文件中不存在、由编译器生成的类成员，引入 Synthetic

属性主要是为了支持嵌套类和嵌套接口。具体细节就不介绍了，感兴趣的读者可以参考 Java 虚拟机规范相关章节。在 ch03\classfile 目录下创建 attr_markers.go 文件，在其中定义 DeprecatedAttribute 和 SyntheticAttribute 结构体，代码如下：

```
package classfile

type DeprecatedAttribute struct { MarkerAttribute }
type SyntheticAttribute struct { MarkerAttribute }

type MarkerAttribute struct{}
func (self *MarkerAttribute) readInfo(reader *ClassReader) {
    // read nothing
}
```

由于这两个属性都没有数据，所以 readInfo() 方法是空的。

3.4.3 SourceFile 属性

SourceFile 是可选定长属性，只会出现在 ClassFile 结构中，用于指出源文件名。其结构定义如下：

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

attribute_length 的值必须是 2。sourcefile_index 是常量池索引，指向 CONSTANT_Utf8_info 常量。在 ch03\classfile 目录下创建 attr_source_file.go 文件，在其中定义 SourceFileAttribute 结构体，代码如下：

```
package classfile

type SourceFileAttribute struct {
    cp              ConstantPool
    sourceFileIndex uint16
}

func (self *SourceFileAttribute) readInfo(reader *ClassReader) {
    self.sourceFileIndex = reader.readUint16()
}
func (self *SourceFileAttribute) FileName() string {
    return self.cp.getUtf8(self.sourceFileIndex)
}
```

笔者的编译器给 ClassFileTest 生成了 SourceFile 属性，如图 3-21 所示。

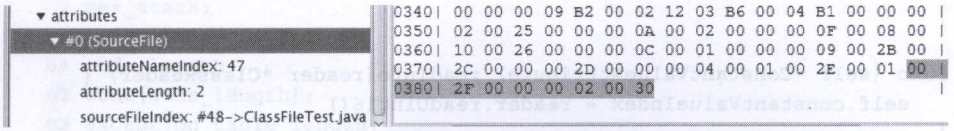


图 3-21　用 classpy 观察 SourceFile 属性

第 47 和第 48 个常量，确实都是 CONSTANT_Utf8_info 常量，如图 3-22 所示。

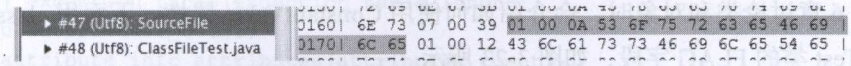


图 3-22　用 classpy 观察 SourceFile 属性（2）

3.4.4　ConstantValue 属性

ConstantValue 是定长属性，只会出现在 field_info 结构中，用于表示常量表达式的值（详见 Java 语言规范的 15.28 节）。其结构定义如下：

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

attribute_length 的值必须是 2。constantvalue_index 是常量池索引，但具体指向哪种常量因字段类型而异。表 3-6 给出了字段类型和常量类型的对应关系。

表 3-5　字段类型和常量类型对应关系

字段类型	常量类型
long	CONSTANT_Long_info
float	CONSTANT_Float_info
double	CONSTANT_Double_info
int, short, char, byte, boolean	CONSTANT_Integer_info
String	CONSTANT_String_info

在 ch03\classfile 目录下创建 attr_constant_value.go 文件，在其中定义 ConstantValueAttribute 结构体，代码如下：

```
package classfile
```

```
type ConstantValueAttribute struct {
    constantValueIndex uint16
}

func (self *ConstantValueAttribute) readInfo(reader *ClassReader) {
    self.constantValueIndex = reader.readUint16()
}
func (self *ConstantValueAttribute) ConstantValueIndex() uint16 {
    return self.constantValueIndex
}
```

在第 6 章讨论类和对象时，会介绍如何使用 ConstantValue 属性。下面用 classpy 观察 ClassFileTest 类的 FLAG 字段，如图 3-23 所示。

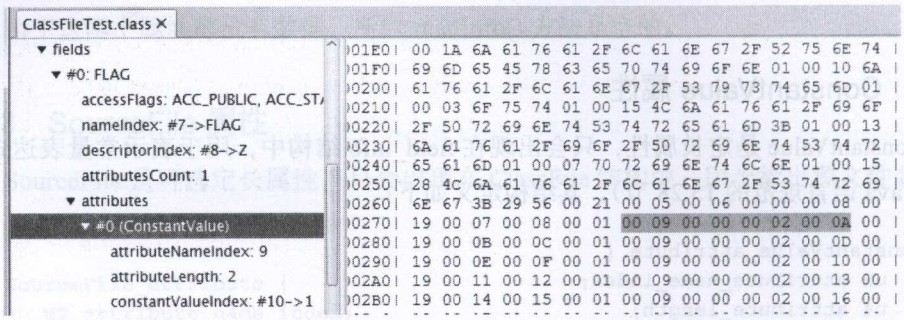


图 3-23 用 classpy 观察 ConstantValue 属性

可以看到，属性表里确实有一个 ConstantValue 属性，constantvalue_index 是 10（0x0A），指向 CONSTANT_Integer_info，如图 3-24 所示。

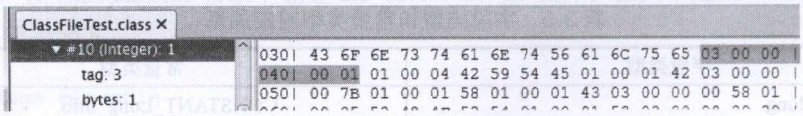


图 3-24 用 classpy 观察 ConstantValue 属性（2）

3.4.5 Code 属性

Code 是变长属性，只存在于 method_info 结构中。Code 属性中存放字节码等方法相关信息。相比前面介绍的几种属性，Code 属性比较复杂，其结构定义如下：

```
Code_attribute {
    u2 attribute_name_index;
```

```
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {   u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

max_stack给出操作数栈的最大深度，max_locals给出局部变量表大小。接着是字节码，存在u1表中。最后是异常处理表和属性表。在第4章讨论运行时数据区，并且实现操作数栈和局部变量表时，max_stack和max_locals就会派上用场。在第5章讨论指令集和解释器时，会用到字节码。在第10章讨论异常处理时，会使用异常处理表。

把Code属性结构翻译成Go结构体，定义在ch03\classfile\attr_code.go文件中，代码如下：

```
package classfile

type CodeAttribute struct {
    cp                  ConstantPool
    maxStack            uint16
    maxLocals           uint16
    code                []byte
    exceptionTable      []*ExceptionTableEntry
    attributes          []AttributeInfo
}

type ExceptionTableEntry struct {
    startPc     uint16
    endPc       uint16
    handlerPc   uint16
    catchType   uint16
}

func (self *CodeAttribute) readInfo(reader *ClassReader) {...}
```

readInfo()方法的代码如下：

```
func (self *CodeAttribute) readInfo(reader *ClassReader) {
```

```
    self.maxStack = reader.readUint16()
    self.maxLocals = reader.readUint16()
    codeLength := reader.readUint32()
    self.code = reader.readBytes(codeLength)
    self.exceptionTable = readExceptionTable(reader)
    self.attributes = readAttributes(reader, self.cp)
}
```

readExceptionTable() 函数的代码如下：

```
func readExceptionTable(reader *ClassReader) []*ExceptionTableEntry {
    exceptionTableLength := reader.readUint16()
    exceptionTable := make([]*ExceptionTableEntry, exceptionTableLength)
    for i := range exceptionTable {
        exceptionTable[i] = &ExceptionTableEntry{
            startPc:   reader.readUint16(),
            endPc:     reader.readUint16(),
            handlerPc: reader.readUint16(),
            catchType: reader.readUint16(),
        }
    }
    return exceptionTable
}
```

ClassFileTest.main() 方法的 Code 属性如图 3-25 所示。

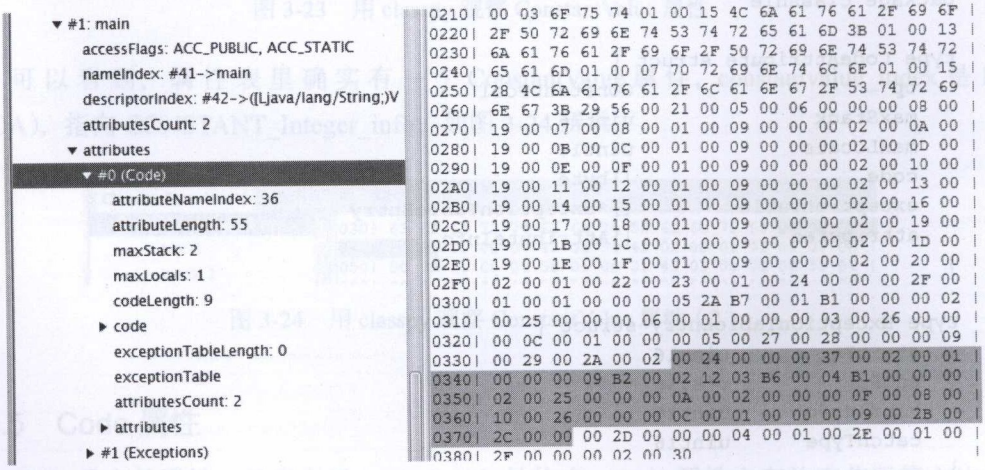


图 3-25　用 classpy 观察 Code 属性

3.4.6　Exceptions 属性

Exceptions 是变长属性，记录方法抛出的异常表，其结构定义如下：

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

在 ch03\classfile 目录下创建 attr_exceptions.go 文件，在其中定义 ExceptionsAttribute 结构体，代码如下：

```
package classfile

type ExceptionsAttribute struct {
    exceptionIndexTable []uint16
}

func (self *ExceptionsAttribute) readInfo(reader *ClassReader) {
    self.exceptionIndexTable = reader.readUint16s()
}
func (self *ExceptionsAttribute) ExceptionIndexTable() []uint16 {
    return self.exceptionIndexTable
}
```

代码比较简单，就不多解释了。ClassFileTest.main() 方法的 Exceptions 属性如图 3-26 所示。

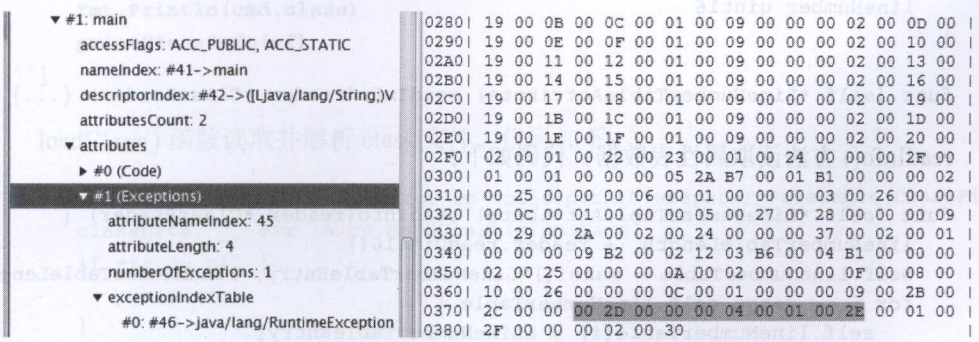


图 3-26　用 classpy 观察 Code 属性

3.4.7　LineNumberTable 和 LocalVariableTable 属性

LineNumberTable 属性表存放方法的行号信息，LocalVariableTable 属性表中存放方法的局部变量信息。这两种属性和前面介绍的 SourceFile 属性都属于调试信息，都不是运行时必需的。在使用 javac 编译器编译 Java 程序时，默认会在 class 文件中生成这些信息。

可以使用 javac 提供的 -g:none 选项来关闭这些信息的生成，这里就不多介绍了，具体请参考 javac 用法。

LineNumberTable 和 LocalVariableTable 属性表在结构上很像，下面以 LineNumberTable 为例进行讨论，它的结构定义如下：

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    {   u2 start_pc;
        u2 line_number;
    } line_number_table[line_number_table_length];
}
```

把上面的结构定义翻译成 Go 结构体，定义在 ch03\classfile\attr_line_number_table.go 文件中，代码如下：

```
package classfile

type LineNumberTableAttribute struct {
    lineNumberTable []*LineNumberTableEntry
}
type LineNumberTableEntry struct {
    startPc    uint16
    lineNumber uint16
}

func (self *LineNumberTableAttribute) readInfo(reader *ClassReader) {...}
```

readInfo() 方法读取属性表数据，代码如下：

```
func (self *LineNumberTableAttribute) readInfo(reader *ClassReader) {
    lineNumberTableLength := reader.readUint16()
    self.lineNumberTable = make([]*LineNumberTableEntry, lineNumberTableLength)
    for i := range self.lineNumberTable {
        self.lineNumberTable[i] = &LineNumberTableEntry{
            startPc:    reader.readUint16(),
            lineNumber: reader.readUint16(),
        }
    }
}
```

在第 10 章讨论异常处理时会详细讨论 LineNumberTable 属性。

3.5　测试本章代码

在第 2 章的测试中，把 class 文件加载到了内存中，并且把一堆看似杂乱无章的数字打印到了控制台。相信读者一定不会满足于此。本节就来修改测试代码，把命令行工具临时打造成一个简化版的 javap。

打开 ch03\main.go 文件，修改 import 语句和 startJVM() 函数，代码如下：

```
package main

import "fmt"
import "strings"
import "jvmgo/ch03/classfile"
import "jvmgo/ch03/classpath"

func main() {...}
func startJVM(options *cmdline.Options, class string, args []string) {...}
```

main() 函数不用变，修改 startJVM() 函数，代码如下：

```
func startJVM(cmd *Cmd) {
    cp := classpath.Parse(cmd.XjreOption, cmd.cpOption)
    className := strings.Replace(cmd.class, ".", "/", -1)
    cf := loadClass(className, cp)
    fmt.Println(cmd.class)
    printClassInfo(cf)
}
```

loadClass() 函数读取并解析 class 文件，代码如下：

```
func loadClass(className string, cp *classpath.Classpath) *classfile.ClassFile {
    classData, _, err := cp.ReadClass(className)
    if err != nil {
        panic(err)
    }
    cf, err := classfile.Parse(classData)
    if err != nil {
        panic(err)
    }
    return cf
}
```

printClassInfo() 函数把 class 文件的一些重要信息打印出来，代码如下：

```
func printClassInfo(cf *classfile.ClassFile) {
    fmt.Printf("version: %v.%v\n", cf.MajorVersion(), cf.MinorVersion())
    fmt.Printf("constants count: %v\n", len(cf.ConstantPool()))
    fmt.Printf("access flags: 0x%x\n", cf.AccessFlags())
    fmt.Printf("this class: %v\n", cf.ClassName())
    fmt.Printf("super class: %v\n", cf.SuperClassName())
    fmt.Printf("interfaces: %v\n", cf.InterfaceNames())
    fmt.Printf("fields count: %v\n", len(cf.Fields()))
    for _, f := range cf.Fields() {
        fmt.Printf("  %s\n", f.Name())
    }
    fmt.Printf("methods count: %v\n", len(cf.Methods()))
    for _, m := range cf.Methods() {
        fmt.Printf("  %s\n", m.Name())
    }
}
```

打开命令行窗口，执行下面的命令编译本章代码。

```
go install jvmgo\ch03
```

编译成功后，在 D:\go\workspace\bin 目录下会出现 ch03.exe 文件。执行 ch03.exe，指定 -Xjre 选项和类名，就可以打印出 class 文件的信息。笔者把 java.lang.String.class 文件（位于 rt.jar 中）的信息打印了出来，如图 3-27 所示。如果读者想测试自己编写的类，记得要指定 -classpath 选项。与第 2 章相比，我们显然取得了很大的进步。

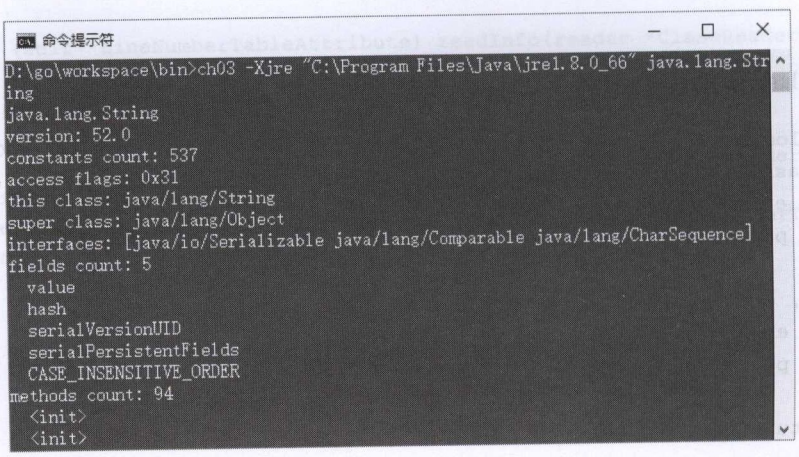
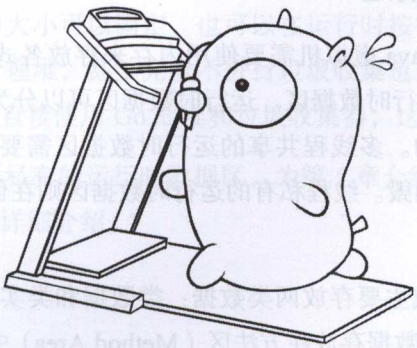


图 3-27　ch03.exe 的测试结果

3.6　本章小结

计算机科学家 David Wheeler 有一句名言：“计算机科学中的任何难题都可以通过增加一个中间层来解决”[⊖]。ClassFile 结构体就是为了实现类加载功能而增加的中间层。在第 6 章，我们会进一步处理 ClassFile 结构体，把它转变 Class 结构体，并放入方法区。不过在此之前，要先在第 4 章实现运行时数据区，在第 5 章实现字节码解释器。

⊖ 原文为 All problems in computer science can be solved by another level of indirection。



第 4 章　运行时数据区

第 1 章编写了命令行工具，第 2 章和第 3 章讨论了如何搜索和解析 class 文件。读者也许有些着急了，为什么读到第 4 章后连 Java 虚拟机的影子都还没有看到？别着急，本章就来讨论并初步实现运行时数据区（run-time data area），为下一章编写字节码解释器做准备。

在开始阅读本章之前，还是先准备好目录结构。复制 ch03 目录，改名为 ch04。修改 main.go 等源文件，把 import 语句中的 ch03 全都改成 ch04，然后在 ch04 目录下创建 rtda [⊖]子目录。现在我们的目录结构应该如下所示：

```
D:\go\workspace\src
  |-jvmgo
    |-ch01 ~ ch03
    |-ch04
      |-classfile
      |-classpath
      |-rtda
      |-cmd.go
      |-main.go
```

⊖ rtda 是 run-time data area 的首字母缩写。

4.1 运行时数据区概述

在运行 Java 程序时，Java 虚拟机需要使用内存来存放各式各样的数据。Java 虚拟机规范把这些内存区域叫作运行时数据区。运行时数据区可以分为两类：一类是多线程共享的，另一类则是线程私有的。多线程共享的运行时数据区需要在 Java 虚拟机启动时创建好，在 Java 虚拟机退出时销毁。线程私有的运行时数据区则在创建线程时才创建，线程退出时销毁。

多线程共享的内存区域主要存放两类数据：类数据和类实例（也就是对象）。对象数据存放在堆（Heap）中，类数据存放在方法区（Method Area）中。堆由垃圾收集器定期清理，所以程序员不需要关心对象空间的释放。类数据包括字段和方法信息、方法的字节码、运行时常量池，等等。从逻辑上来讲，方法区其实也是堆的一部分。

线程私有的运行时数据区用于辅助执行 Java 字节码。每个线程都有自己的 pc 寄存器（Program Counter）和 Java 虚拟机栈（JVM Stack）。Java 虚拟机栈又由栈帧（Stack Frame，后面简称帧）构成，帧中保存方法执行的状态，包括局部变量表（Local Variable）和操作数栈（Operand Stack）等。在任一时刻，某一线程肯定是在执行某个方法。这个方法叫作该线程的当前方法；执行该方法的帧叫作线程的当前帧；声明该方法的类叫作当前类。如果当前方法是 Java 方法，则 pc 寄存器中存放当前正在执行的 Java 虚拟机指令的地址，否则，当前方法是本地方法，pc 寄存器中的值没有明确定义。

根据以上描述，可以大致勾勒出运行时数据区的逻辑结构，如图 4-1 所示。

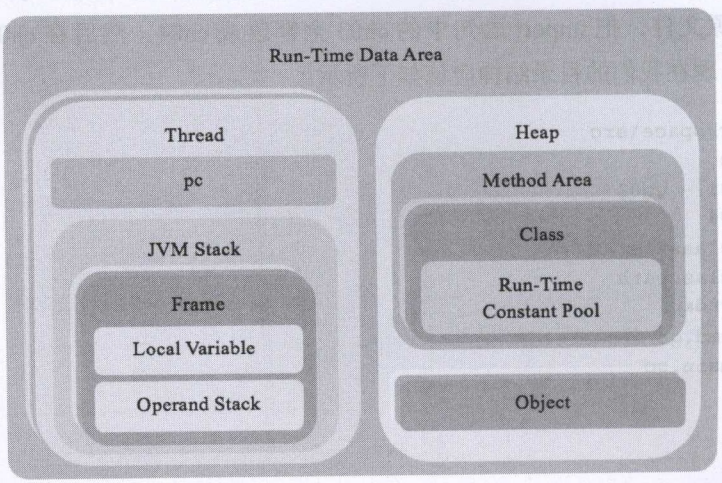


图 4-1 运行时数据区示意图

Java 虚拟机规范对于运行时数据区的规定是相当宽松的。以堆为例：堆可以是连续空间，也可以不连续。堆的大小可以固定，也可以在运行时按需扩展[⊖]。虚拟机实现者可以使用任何垃圾回收算法管理堆，甚至完全不进行垃圾收集也是可以的。由于 Go 本身也有垃圾回收功能，所以可以直接使用 Go 的堆和垃圾收集器，这大大简化了我们的工作。

本章将初步实现线程私有的运行时数据区，为第 5 章介绍指令集打下基础。方法区和运行时常量池将在第 6 章详细介绍。

4.2　数据类型

Java 虚拟机可以操作两类数据：基本类型（primitive type）和引用类型（reference type）。基本类型的变量存放的就是数据本身，引用类型的变量存放的是对象引用，真正的对象数据是在堆里分配的。这里所说的变量包括类变量（静态字段）、实例变量（非静态字段）、数组元素、方法的参数和局部变量，等等。

基本类型可以进一步分为布尔类型（boolean type）和数字类型（numeric type）[⊜]，数字类型又可以分为整数类型（integral type）和浮点数类型（floating-point type）。引用类型可以进一步分为 3 种：类类型、接口类型和数组类型。类类型引用指向类实例，数组类型引用指向数组实例，接口类型引用指向实现了该接口的类或数组实例。引用类型有一个特殊的值——null，表示该引用不指向任何对象。

Go 语言提供了非常丰富的数据类型，包括各种整数和两种精度的浮点数。Java 和 Go 的浮点数都采用 IEEE 754 规范[⊜]。对于基本类型，可以直接在 Go 和 Java 之间建立映射关系。对于引用类型，自然的选择是使用指针。Go 提供了 nil，表示空指针，正好可以用来表示 null。由于要到第 6 章才开始实现类和对象，所以本章先定义一个临时的结构体，用它来表示对象。在 ch04\rtda 目录下创建 object.go，在其中定义 Object 结构体，代码如下：

```
package rtda

type Object struct {
```

⊖ java 命令提供了 -Xms 和 -Xmx 两个非标准选项，用来调整堆的初始大小和最大大小。java 命令的详细介绍请参考第 1 章内容。

⊜ 还有一种基本类型是 returnAddress，它和 jsr、ret、ret_w 指令一起，用来实现 finally 子句。不过从 Java 6 开始，Oracle 的 Java 编译器已经不再使用这三条指令了。详细情况请参考第 10 章内容。

⊜ 本书不讨论浮点数细节，如在内存中的编码形式等。如果读者需要了解这方面的知识，可以阅读 Java 虚拟机规范或 IEEE 754 规范的相关章节。

```
    // todo
}
```

表 4-1 对 Java 虚拟机支持的类型进行了总结。

表 4-1 Java 虚拟机数据类型

Java 数据类型				Go 数据类型
基本类型	数字类型	整数类型	byte	int8
			short	int16
			int	int32
			long	int64
			char	uint16
		浮点数类型	float	float32
			double	float64
	布尔类型		boolean	bool
引用类型			类类型	*Object
			接口类型	*Object
			数组类型	*Object
			null	nil

4.3 实现运行时数据区

前面两节介绍了一些必要的理论，并且定义了 Object 结构体。本节将实现线程私有的运行时数据区。下面先从线程开始。

4.3.1 线程

在 ch04\jvm\rtda 目录下创建 thread.go 文件，在其中定义 Thread 结构体，代码如下：

```
package rtda

type Thread struct {
    pc      int
    stack *Stack
}

func NewThread() *Thread {...}
func (self *Thread) PC() int { return self.pc } // getter
func (self *Thread) SetPC(pc int) { self.pc = pc } // setter
func (self *Thread) PushFrame(frame *Frame) {...}
```

```
func (self *Thread) PopFrame() *Frame {...}
func (self *Thread) CurrentFrame() *Frame {...}
```

目前只定义了 pc 和 stack 两个字段。pc 字段无需解释，stack 字段是 Stack 结构体（Java 虚拟机栈）指针。Stack 结构体在 4.3.2 节介绍。

和堆一样，Java 虚拟机规范对 Java 虚拟机栈的约束也相当宽松。Java 虚拟机栈可以是连续的空间，也可以不连续；可以是固定大小，也可以在运行时动态扩展[⊖]。如果 Java 虚拟机栈有大小限制，且执行线程所需的栈空间超出了这个限制，会导致 StackOverflowError 异常抛出。如果 Java 虚拟机栈可以动态扩展，但是内存已经耗尽，会导致 OutOfMemoryError 异常抛出。

NewThread() 函数创建 Thread 实例的代码如下：

```
func NewThread() *Thread {
    return &Thread{
        stack: newStack(1024),
    }
}
```

newStack() 函数创建 Stack 结构体实例，它的参数表示要创建的 Stack 最多可以容纳多少帧，4.3.2 节将给出这个函数的代码。这里暂时将它赋值为 1024，感兴趣的读者可以修改我们的命令行工具，添加选项来指定这个参数。

PushFrame() 和 PopFrame() 方法只是调用 Stack 结构体的相应方法而已，代码如下：

```
func (self *Thread) PushFrame(frame *Frame) {
    self.stack.push(frame)
}
func (self *Thread) PopFrame() *Frame {
    return self.stack.pop()
}
```

CurrentFrame() 方法返回当前帧，代码如下：

```
func (self *Thread) CurrentFrame() *Frame {
    return self.stack.top()
}
```

4.3.2　Java 虚拟机栈

如前所述，Java 虚拟机规范对 Java 虚拟机栈的约束非常宽松。我们用经典的链表

⊖ java 命令提供了 -Xss 选项来设置 Java 虚拟机栈大小。

（linked list）数据结构来实现 Java 虚拟机栈，这样栈就可以按需使用内存空间，而且弹出的帧也可以及时被 Go 的垃圾收集器回收。在 ch04\jvm\rtda 目录下创建 jvm_stack.go 文件，在其中定义 Stack 结构体，代码如下：

```
package rtda

type Stack struct {
    maxSize      uint
    size         uint
    _top         *Frame
}

func newStack(maxSize uint) *Stack {...}
func (self *Stack) push(frame *Frame) {...}
func (self *Stack) pop() *Frame {...}
func (self *Stack) top() *Frame {...}
```

maxSize 字段保存栈的容量（最多可以容纳多少帧），size 字段保存栈的当前大小，_top 字段保存栈顶指针。newStack() 函数的代码如下：

```
func newStack(maxSize uint) *Stack {
    return &Stack{
        maxSize: maxSize,
    }
}
```

push() 方法把帧推入栈顶，代码如下：

```
func (self *Stack) push(frame *Frame) {
    if self.size >= self.maxSize {
        panic("java.lang.StackOverflowError")
    }
    if self._top != nil {
        frame.lower = self._top
    }
    self._top = frame
    self.size++
}
```

如果栈已经满了，按照 Java 虚拟机规范，应该抛出 StackOverflowError 异常。在第 10 章才会讨论异常，这里先调用 panic() 函数终止程序执行。pop() 方法把栈顶帧弹出，代码如下：

```
func (self *Stack) pop() *Frame {
    if self._top == nil {
```

```
        panic("jvm stack is empty!")
    }

    top := self._top
    self._top = top.lower
    top.lower = nil
    self.size--

    return top
}
```

如果此时栈是空的，肯定是我们的虚拟机有 bug，调用 panic() 函数终止程序执行即可。top() 方法只是返回栈顶帧，但并不弹出，代码如下：

```
func (self *Stack) top() *Frame {
    if self._top == nil {
        panic("jvm stack is empty!")
    }
    return self._top
}
```

4.3.3　帧

在 ch04\jvm\rtda 目录下创建 frame.go 文件，在其中定义 Frame 结构体，代码如下：

```
package rtda

type Frame struct {
    lower          *Frame
    localVars      LocalVars
operandStack       *OperandStack
}

func newFrame(maxLocals, maxStack uint) *Frame {...}
```

Frame 结构体暂时也比较简单，只有三个字段，后续章节还会继续完善它。lower 字段用来实现链表数据结构，localVars 字段保存局部变量表指针，operandStack 字段保存操作数栈指针。NewFrame() 函数创建 Frame 实例，代码如下：

```
func NewFrame(maxLocals, maxStack uint) *Frame {
    return &Frame{
        localVars:    newLocalVars(maxLocals),
        operandStack: newOperandStack(maxStack),
    }
}
```

执行方法所需的局部变量表大小和操作数栈深度是由编译器预先计算好的，存储在 class 文件 method_info 结构的 Code 属性中，具体可以参考 3.4.5 节。

Thread、Stack 和 Frame 结构体的代码都已经给出了，根据代码，可以画出 Java 虚拟机栈的链表结构，如图 4-2 所示。

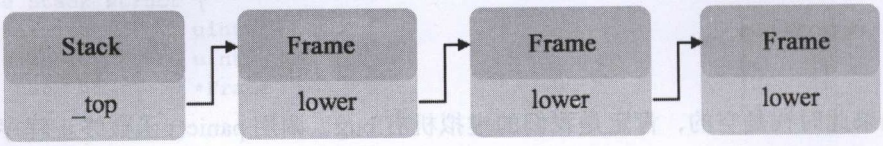


图 4-2 使用链表实现 Java 虚拟机栈

4.3.4 局部变量表

局部变量表是按索引访问的，所以很自然，可以把它想象成一个数组。根据 Java 虚拟机规范，这个数组的每个元素至少可以容纳一个 int 或引用值，两个连续的元素可以容纳一个 long 或 double 值。

那么使用哪种 Go 语言数据类型来表示这个数组呢？最容易想到的是 []int。Go 的 int 类型因平台而异，在 64 位系统上是 int64，在 32 位系统上是 int32，总之足够容纳 Java 的 int 类型。另外它和内置的 uintptr 类型宽度一样，所以也足够放下一个内存地址。通过 unsafe 包可以拿到结构体实例的地址，如下所示：

```
obj := &Object{}
ptr := uintptr(unsafe.Pointer(obj))
ref := int(ptr)
```

但遗憾的是，Go 的垃圾回收机制并不能有效处理 uintptr 指针。也就是说，如果一个结构体实例，除了 uintptr 类型指针保存它的地址之外，其他地方都没有引用这个实例，它就会被当作垃圾回收。

另外一个方案是用 []interface{} 类型，这个方案在实现上没有问题，只是写出来的代码可读性太差。第三种方案是定义一个结构体，让它可以同时容纳一个 int 值和一个引用值。这里将使用第三种方案。在 ch04\rtda 目录下创建 slot.go 文件，在其中定义 Slot 结构体，代码如下：

```
package rtda
```

```
type Slot struct {
    num int32
    ref *Object
}
```

num 字段存放整数，ref 字段存放引用，刚好满足我们的需求。下面用它来实现局部变量表。在 ch04\rtda 目录下创建 local_vars.go 文件，在其中定义 LocalVars 类型，代码如下：

```
package rtda

import "math"

type LocalVars []Slot
```

继续编辑 local_vars.go 文件，在其中定义 newLocalVars() 函数，代码如下：

```
func newLocalVars(maxLocals uint) LocalVars {
    if maxLocals > 0 {
        return make([]Slot, maxLocals)
    }
    return nil
}
```

newLocalVars() 函数创建 LocalVars 实例，代码比较简单，这里就不多解释了。

在第 5 章大家会看到，操作局部变量表和操作数栈的指令都是隐含类型信息的。下面给 LocalVars 类型定义一些方法，用来存取不同类型的变量。int 变量最简单，直接存取即可。

```
func (self LocalVars) SetInt(index uint, val int32) {
    self[index].num = val
}
func (self LocalVars) GetInt(index uint) int32 {
    return self[index].num
}
```

float 变量可以先转成 int 类型，然后按 int 变量来处理。

```
func (self LocalVars) SetFloat(index uint, val float32) {
    bits := math.Float32bits(val)
    self[index].num = int32(bits)
}
func (self LocalVars) GetFloat(index uint) float32 {
    bits := uint32(self[index].num)
    return math.Float32frombits(bits)
}
```

long 变量则需要拆成两个 int 变量。

```
func (self LocalVars) SetLong(index uint, val int64) {
    self[index].num = int32(val)
    self[index+1].num = int32(val >> 32)
}
func (self LocalVars) GetLong(index uint) int64 {
    low := uint32(self[index].num)
    high := uint32(self[index+1].num)
    return int64(high)<<32 | int64(low)
}
```

double 变量可以先转成 long 类型，然后按照 long 变量来处理。

```
func (self LocalVars) SetDouble(index uint, val float64) {
    bits := math.Float64bits(val)
    self.SetLong(index, int64(bits))
}
func (self LocalVars) GetDouble(index uint) float64 {
    bits := uint64(self.GetLong(index))
    return math.Float64frombits(bits)
}
```

最后是引用值，也比较简单，直接存取即可。

```
func (self LocalVars) SetRef(index uint, ref *Object) {
    self[index].ref = ref
}
func (self LocalVars) GetRef(index uint) *Object {
    return self[index].ref
}
```

请读者注意，我们并没有真的对 boolean、byte、short 和 char 类型定义存取方法，这些类型的值都可以转换成 int 值类来处理。下面我们来实现操作数栈。

4.3.5 操作数栈

操作数栈的实现方式和局部变量表类似。在 ch04\rtda 目录下创建 operand_stack.go 文件，在其中定义 OperandStack 结构体，代码如下：

```
package rtda

import "math"

type OperandStack struct {
    size  uint
    slots []Slot
}
```

操作数栈的大小是编译器已经确定的，所以可以用 []Slot 实现。size 字段用于记录栈顶位置。继续编辑 operand_stack.go，在其中实现 newOperandStack() 函数，代码如下：

```
func newOperandStack(maxStack uint) *OperandStack {
    if maxStack > 0 {
        return &OperandStack{
            slots: make([]Slot, maxStack),
        }
    }
    return nil
}
```

代码也比较简单，在此就不多解释了。和局部变量表类似，需要定义一些方法从操作数栈中弹出，或者往其中推入各种类型的变量。先看最简单的 int 变量。

```
func (self *OperandStack) PushInt(val int32) {
    self.slots[self.size].num = val
    self.size++
}
func (self *OperandStack) PopInt() int32 {
    self.size--
    return self.slots[self.size].num
}
```

PushInt() 方法往栈顶放一个 int 变量，然后把 size 加 1。PopInt() 方法则恰好相反，先把 size 减 1，然后返回变量值。float 变量还是先转成 int 类型，然后按 int 变量处理。

```
func (self *OperandStack) PushFloat(val float32) {
    bits := math.Float32bits(val)
    self.slots[self.size].num = int32(bits)
    self.size++
}
func (self *OperandStack) PopFloat() float32 {
    self.size--
    bits := uint32(self.slots[self.size].num)
    return math.Float32frombits(bits)
}
```

把 long 变量推入栈顶时，要拆成两个 int 变量。弹出时，先弹出两个 int 变量，然后组装成一个 long 变量。

```
func (self *OperandStack) PushLong(val int64) {
    self.slots[self.size].num = int32(val)
    self.slots[self.size+1].num = int32(val >> 32)
    self.size += 2
}
```

```
func (self *OperandStack) PopLong() int64 {
    self.size -= 2
    low := uint32(self.slots[self.size].num)
    high := uint32(self.slots[self.size+1].num)
    return int64(high)<<32 | int64(low)
}
```

double 变量先转成 long 类型，然后按 long 变量处理。

```
func (self *OperandStack) PushDouble(val float64) {
    bits := math.Float64bits(val)
    self.PushLong(int64(bits))
}
func (self *OperandStack) PopDouble() float64 {
    bits := uint64(self.PopLong())
    return math.Float64frombits(bits)
}
```

最后看引用类型，代码如下：

```
func (self *OperandStack) PushRef(ref *Object) {
    self.slots[self.size].ref = ref
    self.size++
}
func (self *OperandStack) PopRef() *Object {
    self.size--
    ref := self.slots[self.size].ref
    self.slots[self.size].ref = nil
    return ref
}
```

PushRef() 方法比较简单，此处不做太多解释。PopRef() 方法需要说明一点：弹出引用后，把 Slot 结构体的 ref 字段设置成 nil，这样做是为了帮助 Go 的垃圾收集器回收 Object 结构体实例。

至此，局部变量表和操作数栈都准备好了。仅通过代码来理解它们可能不是很直观，下面我们将通过一个具体的例子来分析局部变量表和操作数的使用。

4.3.6 局部变量表和操作数栈实例分析

以圆形的周长公式为例进行分析，下面是 Java 方法的代码。

```
public static float circumference(float r) {
    float pi = 3.14f;
    float area = 2 * pi * r;
```

```
    return area;
}
```

上面的方法会被 javac 编译器编译成如下字节码：

```
00  ldc #4
02  fstore_1
03  fconst_2
04  fload_1
05  fmul
06  fload_0
07  fmul
08  fstore_2
09  fload_2
10  return
```

下面分析这段字节码的执行。circumference() 方法的局部变量表大小是 3，操作数栈深度是 2。假设调用方法时，传递给它的参数是 1.6f，方法开始执行前，帧的状态如图 4-3 所示。

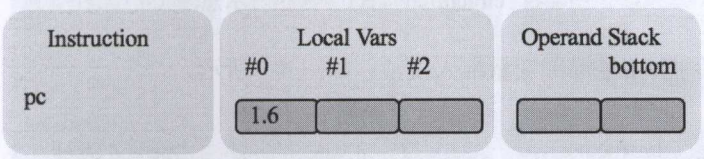


图 4-3　circumference() 方法执行示意图（1）

第一条指令是 ldc，它把 3.14f 推入栈顶，如图 4-4 所示。

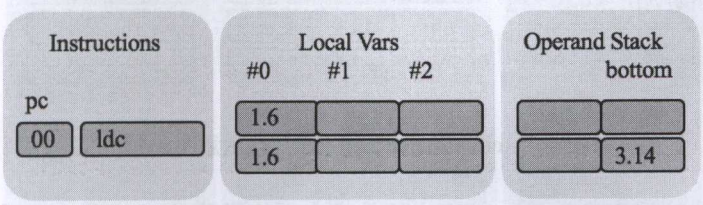


图 4-4　circumference() 方法执行示意图（2）

注意，上面是局部变量表和操作数栈过去的状态，最下面是当前状态。接着是 fstore_1 指令，它把栈顶的 3.14f 弹出，放到 #1 号局部变量中，如图 4-5 所示。

fconst_2 指令把 2.0f 推到栈顶，如图 4-6 所示。

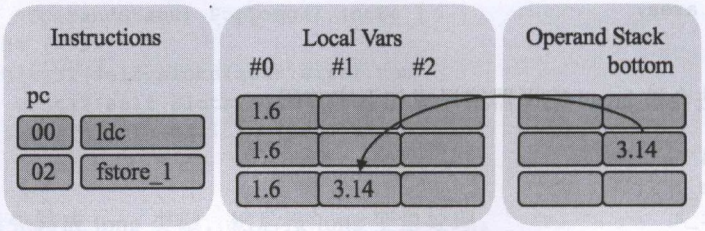


图 4-5 circumference() 方法执行示意图（3）

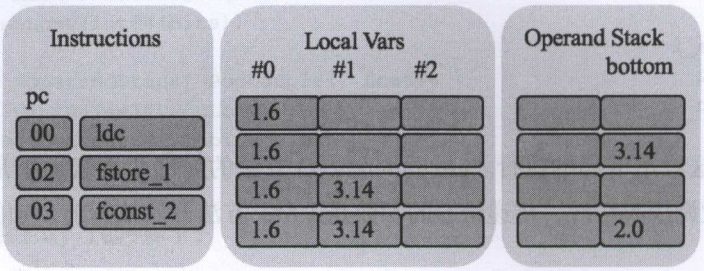


图 4-6 circumference() 方法执行示意图（4）

fload_1 指令把 #1 号局部变量推入栈顶，如图 4-7 所示。

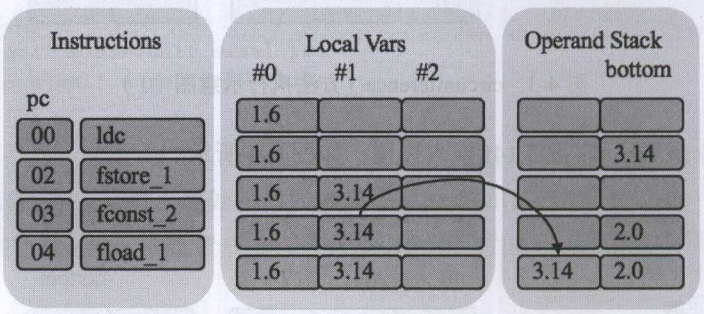


图 4-7 circumference() 方法执行示意图（5）

fmul 指令执行浮点数乘法。它把栈顶的两个浮点数弹出，相乘，然后把结果推入栈顶，如图 4-8 所示。

fload_0 指令把 #0 号局部变量推入栈顶，如图 4-9 所示。

fmul 继续乘法计算，如图 4-10 所示。

fstore_2 指令把操作数栈顶的 float 值弹出，放入 #2 号局部变量表，如图 4-11 所示。

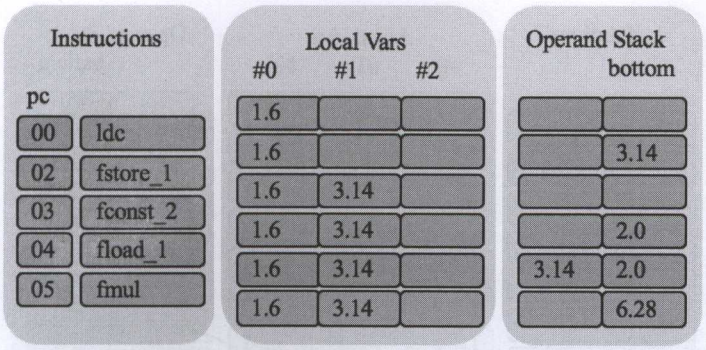


图 4-8　circumference() 方法执行示意图（6）

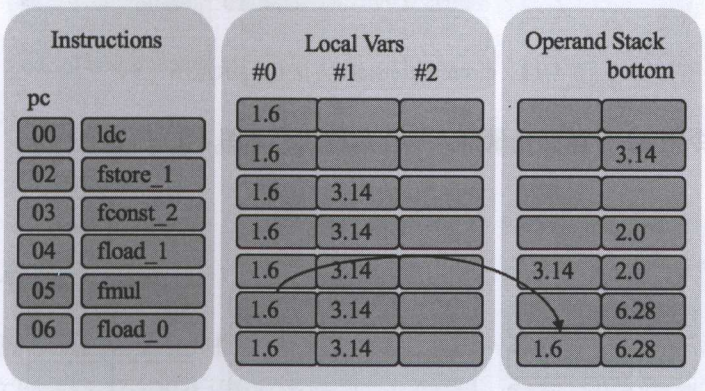


图 4-9　circumference() 方法执行示意图（7）

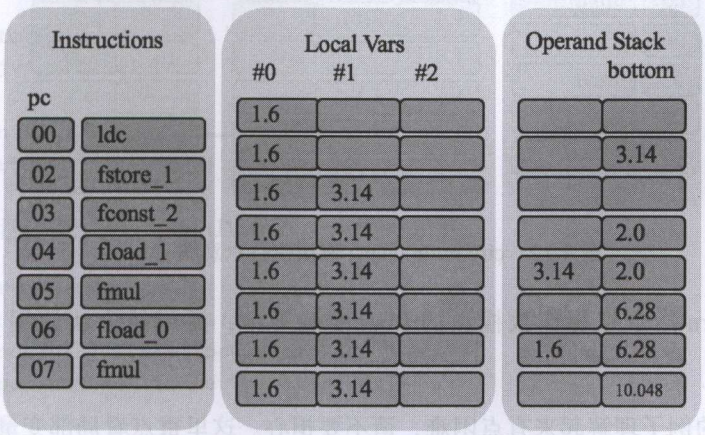


图 4-10　circumference() 方法执行示意图（8）

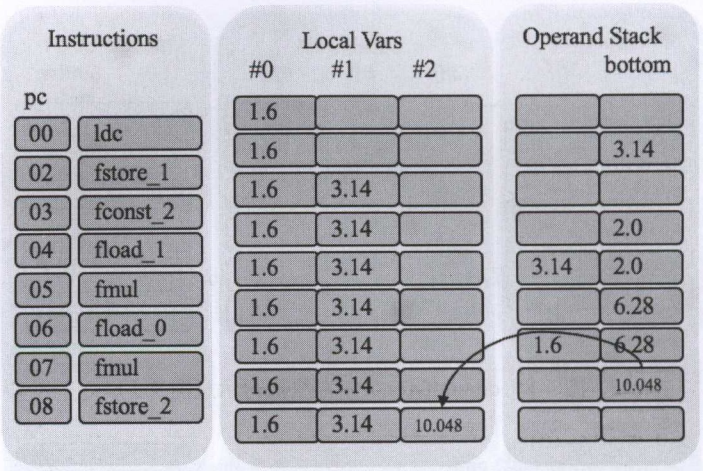


图 4-11 circumference() 方法执行示意图（9）

fload_2 指令把 #2 号局部变量推入操作数栈顶，如图 4-12 所示。

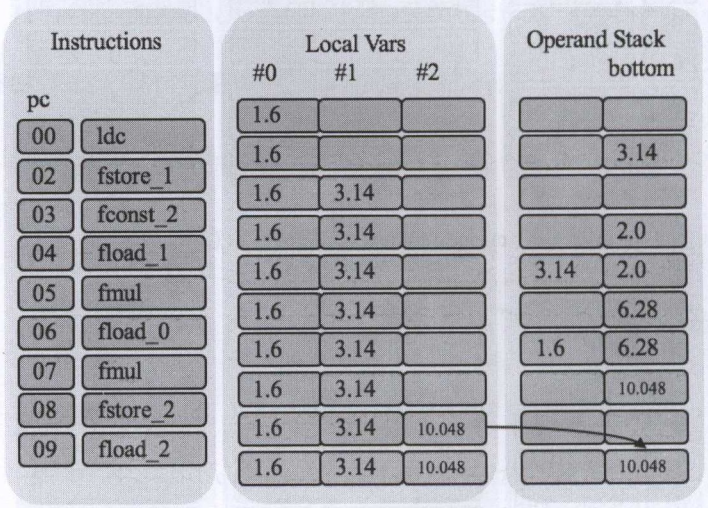


图 4-12 circumference() 方法执行示意图（10）

最后 freturn 指令把操作数栈顶的 float 变量弹出，返回给方法调用者，如图 4-13 所示。

如果上面的例子理解起来有点困难，请不要担心。这里重点看局部变量表和操作数栈的用法就可以了，后面的章节会详细介绍 Java 虚拟机指令集。

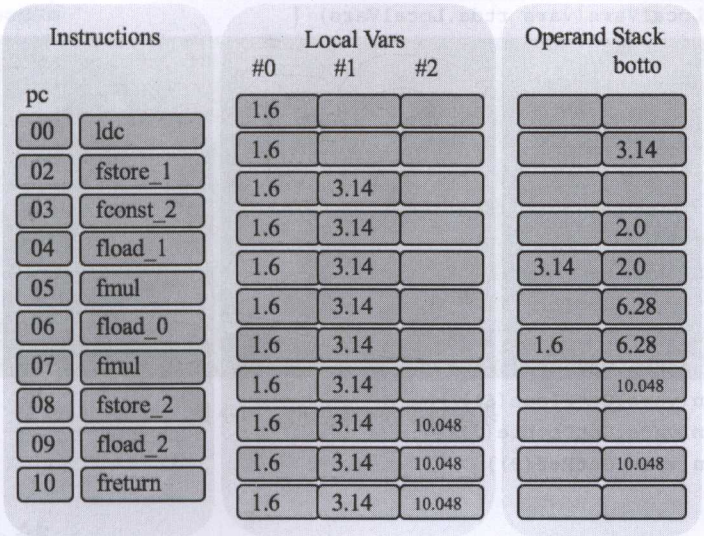


图 4-13　circumference() 方法执行示意图（11）

4.4　测试本章代码

本节简单测试局部变量表和操作数栈的用法，在下一章它们才会真正派上用场。打开 ch04\main.go 文件，修改 import 语句，代码如下：

```
package main

import "fmt"
import "jvmgo/ch04/rtda"

func main() {...}
func startJVM(cmd *Cmd) {...}
```

main() 方法不变，修改 startJVM 方法，代码如下：

```
func startJVM(cmd *Cmd) {
    frame := rtda.NewFrame(100, 100)
    testLocalVars(frame.LocalVars())
    testOperandStack(frame.OperandStack())
}
```

testLocalVars() 函数测试局部变量，代码如下：

```
func testLocalVars(vars rtda.LocalVars) {
    vars.SetInt(0, 100)
    vars.SetInt(1, -100)
    vars.SetLong(2, 2997924580)
    vars.SetLong(4, -2997924580)
    vars.SetFloat(6, 3.1415926)
    vars.SetDouble(7, 2.71828182845)
    vars.SetRef(9, nil)
    println(vars.GetInt(0))
    println(vars.GetInt(1))
    println(vars.GetLong(2))
    println(vars.GetLong(4))
    println(vars.GetFloat(6))
    println(vars.GetDouble(7))
    println(vars.GetRef(9))
}
```

testOperandStack() 函数测试操作数栈，代码如下：

```
func testOperandStack(ops *rtda.OperandStack) {
    ops.PushInt(100)
    ops.PushInt(-100)
    ops.PushLong(2997924580)
    ops.PushLong(-2997924580)
    ops.PushFloat(3.1415926)
    ops.PushDouble(2.71828182845)
    ops.PushRef(nil)
    println(ops.PopRef())
    println(ops.PopDouble())
    println(ops.PopFloat())
    println(ops.PopLong())
    println(ops.PopLong())
    println(ops.PopInt())
    println(ops.PopInt())
}
```

打开命令行窗口，执行下面的命令编译本章代码：

```
go install jvmgo\ch04
```

编译成功后，在 D:\go\workspace\bin 目录下出现 ch04.exe 文件。执行 ch04.exe，运行结果如图 4-14 所示。

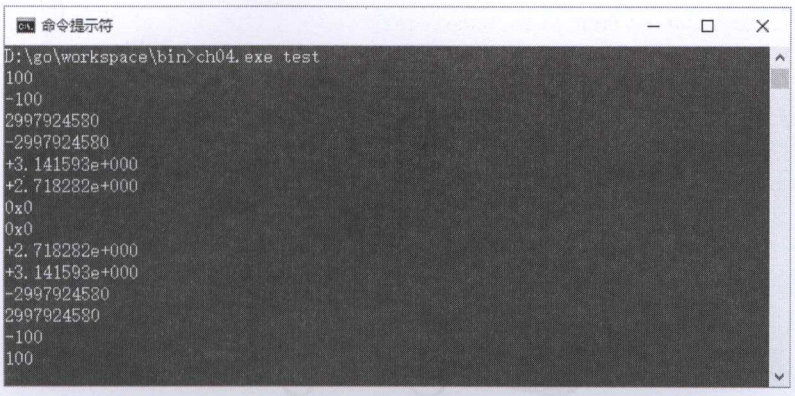
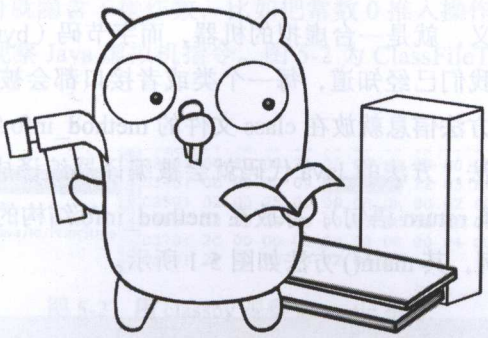


图 4-14　ch04.exe 的运行结果

4.5　本章小结

本章介绍了运行时数据区，初步实现了 Thread、Stack、Frame、OperandStack 和 LocalVars 等线程私有的运行时数据区。下一章将实现字节码解释器，到时候方法就可以在我们的 Java 虚拟机里运行了。



第 5 章　指令集和解释器

由第 3 章可知，编译之后的 Java 方法以字节码的形式存储在 class 文件中。在第 4 章中，初步实现了 Java 虚拟机栈、帧、操作数栈和局部变量表等运行时数据区。本章将在前两章的基础上编写一个简单的解释器，并且实现大约 150 条指令。在后面的章节中，会不断改进这个解释器，让它可以执行更多的指令。

在开始阅读本章之前，先把本章的目录结构准备好。复制 ch04 目录，改名为 ch05。修改 main.go 等文件，把 import 语句中的 ch04 都改成 ch05。在 ch05 目录中创建 instructions 子目录。现在我们的目录结构看起来应该是下面这样：

```
D:\go\workspace\src
|-jvmgo
  |-ch01 ~ ch04
  |-ch05
    |-classfile
    |-classpath
    |-instructions
    |-rtda
    |-cmd.go
    |-main.go
```

5.1 字节码和指令集

Java 虚拟机顾名思义，就是一台虚拟的机器，而字节码（bytecode）就是运行在这台虚拟机器上的机器码。我们已经知道，每一个类或者接口都会被 Java 编译器编译成一个 class 文件，类或接口的方法信息就放在 class 文件的 method_info 结构中[⊖]。如果方法不是抽象的，也不是本地方法，方法的 Java 代码就会被编译器编译成字节码（即使方法是空的，编译器也会生成一条 return 语句），存放在 method_info 结构的 Code 属性中。仍以第 3 章的 ClassFileTest 类为例，其 main() 方法如图 5-1 所示。

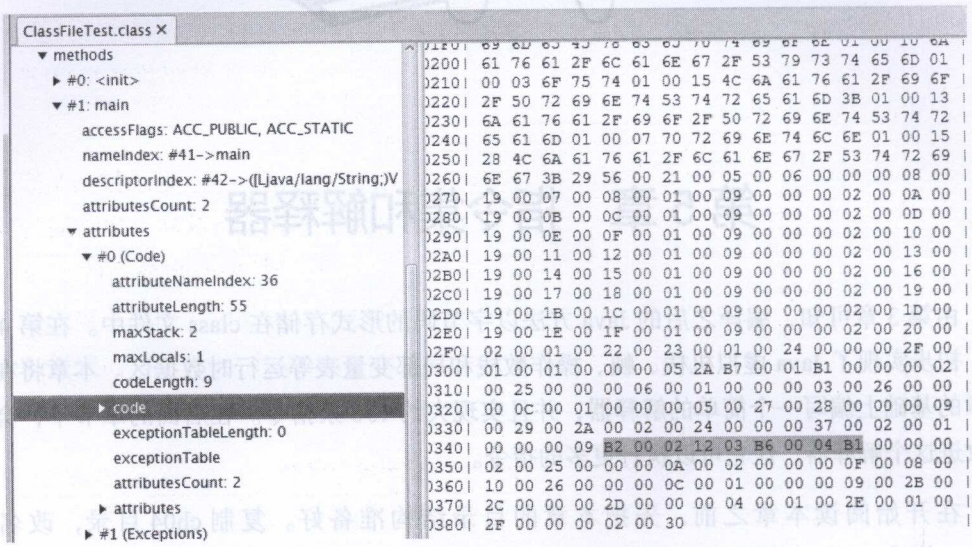


图 5-1 用 classpy 观察方法字节码

字节码中存放编码后的 Java 虚拟机指令。每条指令都以一个单字节的操作码（opcode）开头，这就是字节码名称的由来。由于只使用一字节表示操作码，显而易见，Java 虚拟机最多只能支持 256（2^8）条指令。到第八版为止，Java 虚拟机规范已经定义了 205 条指令，操作码分别是 0（0x00）到 202（0xCA）、254（0xFE）和 255（0xFF）。这 205 条指令构成了 Java 虚拟机的指令集（instruction set）。和汇编语言类似，为了便于记忆，Java 虚拟机规范给每个操作码都指定了一个助记符（mnemonic）。比如操作码是 0x00 这条指令，因为它什么也不做，所以它的助记符是 nop（no operation）。

Java 虚拟机使用的是变长指令，操作码后面可以跟零字节或多字节的操作数

⊖ 如果读者已经忘记了 class 文件结构，可以回到第 3 章复习。

（operand）。如果把指令想象成函数的话，操作数就是它的参数。为了让编码后的字节码更加紧凑，很多操作码本身就隐含了操作数，比如把常数 0 推入操作数栈的指令是 iconst_0。下面通过具体的例子来观察 Java 虚拟机指令。图 5-2 为 ClassFileTest.main() 方法的第一条指令。

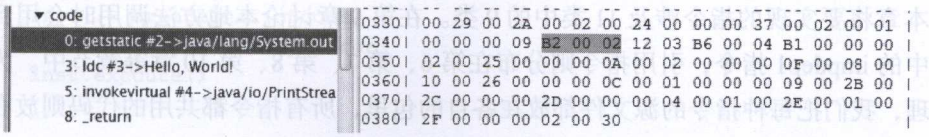


图 5-2　用 classpy 观察 getstatic 指令

可以看到，该指令的操作码是 0xB2，助记符是 getstatic。它的操作数是 0x0002，代表常量池里的第二个常量。

在第 4 章中讨论过，操作数栈和局部变量表只存放数据的值，并不记录数据类型。结果就是：指令必须知道自己在操作什么类型的数据。这一点也直接反映在了操作码的助记符上。例如，iadd 指令就是对 int 值进行加法操作；dstore 指令把操作数栈顶的 double 值弹出，存储到局部变量表中；areturn 从方法中返回引用值。也就是说，如果某类指令可以操作不同类型的变量，则助记符的第一个字母表示变量类型。助记符首字母和变量类型的对应关系如表 5-1 所示。

表 5-1　助记符首字母和变量类型对应表

助记符首字母	数据类型	例　子
a	reference	aload、astore、areturn
b	byte/boolean	bipush、baload
c	char	caload、castore
d	double	dload、dstore、dadd
f	float	fload、fstore、fadd
i	int	iload、istore、iadd
l	long	load、lstore、ladd
s	short	sipush、sastore

Java 虚拟机规范把已经定义的 205 条指令按用途分成了 11 类，分别是：常量（constants）指令、加载（loads）指令、存储（stores）指令、操作数栈（stack）指令、数学（math）指令、转换（conversions）指令、比较（comparisons）指令、控制（control）指令、引用（references）指令、扩展（extended）指令和保留（reserved）指令。

保留指令一共有 3 条。其中一条是留给调试器的，用于实现断点，操作码是 202（0xCA），助记符是 breakpoint。另外两条留给 Java 虚拟机实现内部使用，操作码分别是 254（0xFE）和 266（0xFF），助记符是 impdep1 和 impdep2。这三条指令不允许出现在 class 文件中。

本章将要实现的指令涉及 11 类中的 9 类。在第 9 章讨论本地方法调用时会用到保留指令中的 impdep1 指令，引用指令则分布在第 6、第 7、第 8、第 10 章等章节中。为了便于管理，我们把每种指令的源文件都放在各自的包里，所有指令都共用的代码则放在 base 包里。因此 ch05\instructions 目录下会有如下 10 个子目录：

```
D:\go\workspace\src
  |-jvmgo
    |-ch05
      |-classfile
      |-classpath
      |-rtda
      |-instructions
        |-base
        |-comparisons
        |-constants
        |-control
        |-conversions
        |-extended
        |-loads
        |-math
        |-stack
        |-stores
```

请读者先创建好这些目录。

5.2 指令和指令解码

Java 虚拟机规范的 2.11 节介绍了 Java 虚拟机解释器的大致逻辑，如下所示：

```
do {
   atomically calculate pc and fetch opcode at pc;
   if (operands) fetch operands;
   execute the action for the opcode;
} while (there is more to do);
```

每次循环都包含三个部分：计算 pc、指令解码、指令执行。可以把这个逻辑用 Go 语言写成一个 for 循环，里面是个大大的 switch-case 语句。但这样的话，代码的可读性将非

常差。所以采用另外一种方式：把指令抽象成接口，解码和执行逻辑写在具体的指令实现中。这样编写出的解释器就和 Java 虚拟机规范里的伪代码一样简单，代码如下：

```
for {
    pc := calculatePC()
    opcode := bytecode[pc]
    inst := createInst(opcode)
    inst.fetchOperands(bytecode)
    inst.execute()
}
```

上面给出的仍然是伪代码。将在 5.12 节编写解释器代码，在 5.3 ~ 5.11 节分类实现具体的指令。本节先定义指令接口，然后定义一个结构体用来辅助指令解码。

5.2.1　Instruction 接口

在 ch05\instructions\base 目录下创建 instruction.go 文件，在其中定义 Instruction 接口，代码如下：

```
package base

import "jvmgo/ch05/rtda"

type Instruction interface {
    FetchOperands(reader *BytecodeReader)
    Execute(frame *rtda.Frame)
}
```

FetchOperands() 方法从字节码中提取操作数，Execute() 方法执行指令逻辑。有很多指令的操作数都是类似的。为了避免重复代码，按照操作数类型定义一些结构体，并实现 FetchOperands() 方法。这相当于 Java 中的抽象类，具体的指令继承这些结构体，然后专注实现 Execute() 方法即可。

在 instruction.go 文件中定义 NoOperandsInstruction 结构体，代码如下：

```
type NoOperandsInstruction struct {}
```

NoOperandsInstruction 表示没有操作数的指令，所以没有定义任何字段。FetchOperands() 方法自然也是空空如也，什么也不用读，代码如下：

```
func (self *NoOperandsInstruction) FetchOperands(reader *BytecodeReader) {
    // nothing to do
}
```

继续编辑 instruction.go 文件，在其中定义 BranchInstruction 结构体，代码如下：

```
type BranchInstruction struct {
    Offset int
}
```

BranchInstruction 表示跳转指令，Offset 字段存放跳转偏移量。FetchOperands() 方法从字节码中读取一个 uint16 整数，转成 int 后赋给 Offset 字段。代码如下：

```
func (self *BranchInstruction) FetchOperands(reader *BytecodeReader) {
    self.Offset = int(reader.ReadInt16())
}
```

继续编辑 instruction.go 文件，在其中定义 Index8Instruction 结构体，代码如下：

```
type Index8Instruction struct {
    Index uint
}
```

存储和加载类指令需要根据索引存取局部变量表，索引由单字节操作数给出。把这类指令抽象成 Index8Instruction 结构体，用 Index 字段表示局部变量表索引。FetchOperands() 方法从字节码中读取一个 int8 整数，转成 uint 后赋给 Index 字段。代码如下：

```
func (self *Index8Instruction) FetchOperands(reader *BytecodeReader) {
    self.Index = uint(reader.ReadUint8())
}
```

最后在 instruction.go 文件中定义 Index16Instruction 结构体，代码如下：

```
type Index16Instruction struct {
    Index uint
}
```

有一些指令需要访问运行时常量池，常量池索引由两字节操作数给出。把这类指令抽象成 Index16Instruction 结构体，用 Index 字段表示常量池索引。FetchOperands() 方法从字节码中读取一个 uint16 整数，转成 uint 后赋给 Index 字段。代码如下：

```
func (self *Index16Instruction) FetchOperands(reader *BytecodeReader) {
    self.Index = uint(reader.ReadUint16())
}
```

指令接口和“抽象”指令定义好了，下面来看 BytecodeReader 结构体。

5.2.2　BytecodeReader

在 ch05\instructions\base 目录下创建 bytecode_reader.go 文件，在其中定义 BytecodeReader 结构体，代码如下：

```
package base

type BytecodeReader struct {
    code    []byte
    pc      int
}
```

code 字段存放字节码，pc 字段记录读取到了哪个字节。为了避免每次解码指令都新创建一个 BytecodeReader 实例，给它定义一个 Reset() 方法，代码如下：

```
func (self *BytecodeReader) Reset(code []byte, pc int) {
    self.code = code
    self.pc = pc
}
```

下面实现一系列的 Read() 方法。先看最简单的 ReadUint8() 方法，代码如下：

```
func (self *BytecodeReader) ReadUint8() uint8 {
    i := self.code[self.pc]
    self.pc++
    return i
}
```

ReadInt8() 方法调用 ReadUint8()，然后把读取到的值转成 int8 返回，代码如下：

```
func (self *BytecodeReader) ReadInt8() int8 {
    return int8(self.ReadUint8())
}
```

ReadUint16() 连续读取两字节，代码如下：

```
func (self *BytecodeReader) ReadUint16() uint16 {
    byte1 := uint16(self.ReadUint8())
    byte2 := uint16(self.ReadUint8())
    return (byte1 << 8) | byte2
}
```

ReadInt16() 方法调用 ReadUint16()，然后把读取到的值转成 int16 返回，代码如下：

```
func (self *BytecodeReader) ReadInt16() int16 {
    return int16(self.ReadUint16())
}
```

ReadInt32() 方法连续读取 4 字节，代码如下：

```
func (self *BytecodeReader) ReadInt32() int32 {
    byte1 := int32(self.ReadUint8())
    byte2 := int32(self.ReadUint8())
    byte3 := int32(self.ReadUint8())
    byte4 := int32(self.ReadUint8())
    return (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4
}
```

还需要定义两个方法：ReadInt32s() 和 SkipPadding()。这两个方法只有 tableswitch 和 lookupswitch 指令使用，介绍这两条指令时再给出代码。

在接下来的 9 个小节中，将按照分类依次实现约 150 条指令，占整个指令集的 3/4。读者千万不要被 150 这个数字吓倒，因为很多指令其实是非常相似的。比如 iload、lload、fload、dload 和 aload 这 5 条指令，除了操作的数据类型不同以外，代码几乎一样。再比如 iload_0、iload_1、iload_2 和 iload_3 这四条指令，只是 iload 指令的特例（局部变量表索引隐含在操作码中），操作逻辑完全一样。

如果逐一列出这 150 余条指令的代码，既枯燥乏味，也相当浪费纸张。为了节约篇幅，只讨论一些具有代表意义的指令的实现代码，从这些指令可以很容易想象到其他指令的实现。附录 A 给出了完整的指令集列表，里面有每个指令的操作码、助记符和本书中实现它们的章节，以方便读者参考。

5.3 常量指令

常量指令把常量推入操作数栈顶。常量可以来自三个地方：隐含在操作码里、操作数和运行时常量池。常量指令共有 21 条，本节实现其中的 18 条。另外 3 条是 ldc 系列指令，用于从运行时常量池中加载常量，将在第 6 章介绍。

5.3.1 nop 指令

nop 指令是最简单的一条指令，因为它什么也不做。在 ch05\instructions\constants 目录下创建 nop.go 文件，在其中实现 nop 指令，代码如下：

```
package constants

import "jvmgo/ch05/instructions/base"
```

```
import "jvmgo/ch05/rtda"

// Do nothing
type NOP struct{ base.NoOperandsInstruction }

func (self *NOP) Execute(frame *rtda.Frame) {
    // 什么也不用做
}
```

5.3.2　const 系列指令

这一系列指令把隐含在操作码中的常量值推入操作数栈顶。在 ch05\instructions\constants 目录下创建 const.go 文件，在其中定义 15 条指令，代码如下：

```
package constants

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type ACONST_NULL struct{ base.NoOperandsInstruction }
type DCONST_0 struct{ base.NoOperandsInstruction }
type DCONST_1 struct{ base.NoOperandsInstruction }
type FCONST_0 struct{ base.NoOperandsInstruction }
type FCONST_1 struct{ base.NoOperandsInstruction }
type FCONST_2 struct{ base.NoOperandsInstruction }
type ICONST_M1 struct{ base.NoOperandsInstruction }
type ICONST_0 struct{ base.NoOperandsInstruction }
type ICONST_1 struct{ base.NoOperandsInstruction }
type ICONST_2 struct{ base.NoOperandsInstruction }
type ICONST_3 struct{ base.NoOperandsInstruction }
type ICONST_4 struct{ base.NoOperandsInstruction }
type ICONST_5 struct{ base.NoOperandsInstruction }
type LCONST_0 struct{ base.NoOperandsInstruction }
type LCONST_1 struct{ base.NoOperandsInstruction }
```

以 3 条指令为例进行说明。aconst_null 指令把 null 引用推入操作数栈顶，代码如下：

```
func (self *ACONST_NULL) Execute(frame *rtda.Frame) {
    frame.OperandStack().PushRef(nil)
}
```

dconst_0 指令把 double 型 0 推入操作数栈顶，代码如下：

```
func (self *DCONST_0) Execute(frame *rtda.Frame) {
    frame.OperandStack().PushDouble(0.0)
}
```

iconst_m1 指令把 int 型 -1 推入操作数栈顶，代码如下：

```
func (self *ICONST_M1) Execute(frame *rtda.Frame) {
    frame.OperandStack().PushInt(-1)
}
```

5.3.3 bipush 和 sipush 指令

bipush 指令从操作数中获取一个 byte 型整数，扩展成 int 型，然后推入栈顶。sipush 指令从操作数中获取一个 short 型整数，扩展成 int 型，然后推入栈顶。在 ch05\instructions\constants 目录下创建 ipush.go 文件，在其中定义 bipush 和 sipush 指令，代码如下：

```
package constants

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type BIPUSH struct { val int8 } // Push byte
type SIPUSH struct { val int16 } // Push short
```

以 bipush 指令为例，FetchOperands() 和 Execute() 方法的代码如下：

```
func (self *BIPUSH) FetchOperands(reader *base.BytecodeReader) {
    self.val = reader.ReadInt8()
}
func (self *BIPUSH) Execute(frame *rtda.Frame) {
    i := int32(self.val)
    frame.OperandStack().PushInt(i)
}
```

5.4 加载指令

加载指令从局部变量表获取变量，然后推入操作数栈顶。加载指令共 33 条，按照所操作变量的类型可以分为 6 类：aload 系列指令操作引用类型变量、dload 系列操作 double 类型变量、fload 系列操作 float 变量、iload 系列操作 int 变量、lload 系列操作 long 变量、xaload 操作数组。本节实现其中的 25 条，数组和 xaload 系列指令将在第 8 章讨论。下面以 iload 系列为例介绍加载指令。

在 ch05\instructions\loads 目录下创建 iload.go 文件，在其中定义 5 条指令，代码如下：

```
package loads
```

```
import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Load int from local variable
type ILOAD struct{ base.Index8Instruction }
type ILOAD_0 struct{ base.NoOperandsInstruction }
type ILOAD_1 struct{ base.NoOperandsInstruction }
type ILOAD_2 struct{ base.NoOperandsInstruction }
type ILOAD_3 struct{ base.NoOperandsInstruction }
```

为了避免重复代码，定义一个函数供 iload 系列指令使用，代码如下：

```
func _iload(frame *rtda.Frame, index uint) {
    val := frame.LocalVars().GetInt(index)
    frame.OperandStack().PushInt(val)
}
```

iload 指令的索引来自操作数，其 Execute() 方法如下：

```
func (self *ILOAD) Execute(frame *rtda.Frame) {
    _iload(frame, uint(self.Index))
}
```

其余 4 条指令的索引隐含在操作码中，以 iload_1 为例，其 Execute() 方法如下：

```
func (self *ILOAD_1) Execute(frame *rtda.Frame) {
    _iload(frame, 1)
}
```

5.5　存储指令

和加载指令刚好相反，存储指令把变量从操作数栈顶弹出，然后存入局部变量表。和加载指令一样，存储指令也可以分为 6 类。以 lstore 系列指令为例进行介绍。在 ch05\instructions\stores 目录下创建 lstore.go 文件，在其中定义 5 条指令，代码如下：

```
package stores

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Store long into local variable
type LSTORE struct{ base.Index8Instruction }
type LSTORE_0 struct{ base.NoOperandsInstruction }
type LSTORE_1 struct{ base.NoOperandsInstruction }
type LSTORE_2 struct{ base.NoOperandsInstruction }
```

```
type LSTORE_3 struct{ base.NoOperandsInstruction }
```

同样定义一个函数供 5 条指令使用，代码如下：

```
func _lstore(frame *rtda.Frame, index uint) {
    val := frame.OperandStack().PopLong()
    frame.LocalVars().SetLong(index, val)
}
```

lstore 指令的索引来自操作数，其 Execute() 方法如下：

```
func (self *LSTORE) Execute(frame *rtda.Frame) {
    _lstore(frame, uint(self.Index))
}
```

其余 4 条指令的索引隐含在操作码中，以 lstore_2 为例，其 Execute() 方法如下：

```
func (self *LSTORE_2) Execute(frame *rtda.Frame) {
    _lstore(frame, 2)
}
```

5.6 栈指令

栈指令直接对操作数栈进行操作，共 9 条：pop 和 pop2 指令将栈顶变量弹出，dup 系列指令复制栈顶变量，swap 指令交换栈顶的两个变量。

和其他类型的指令不同，栈指令并不关心变量类型。为了实现栈指令，需要给 OperandStack 结构体添加两个方法。打开 ch05\rtda\operand_stack.go 文件，在其中定义 PushSlot() 和 PopSlot() 方法，代码如下：

```
func (self *OperandStack) PushSlot(slot Slot) {
    self.slots[self.size] = slot
    self.size++
}
func (self *OperandStack) PopSlot() Slot {
    self.size--
    return self.slots[self.size]
}
```

5.6.1 pop 和 pop2 指令

在 ch05\instructions\stack 目录下创建 pop.go 文件，在其中定义 pop 和 pop2 指令，代

码如下：

```
package stack
import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type POP struct{ base.NoOperandsInstruction }
type POP2 struct{ base.NoOperandsInstruction }
```

pop 指令把栈顶变量弹出，代码如下：

```
func (self *POP) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    stack.PopSlot()
}
```

pop 指令只能用于弹出 int、float 等占用一个操作数栈位置的变量。double 和 long 变量在操作数栈中占据两个位置，需要使用 pop2 指令弹出，代码如下：

```
func (self *POP2) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    stack.PopSlot()
    stack.PopSlot()
}
```

5.6.2　dup 指令

在 ch05\instructions\stack 目录下创建 dup.go 文件，在其中定义 6 条指令，代码如下：

```
package stack

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type DUP struct{ base.NoOperandsInstruction }
type DUP_X1 struct{ base.NoOperandsInstruction }
type DUP_X2 struct{ base.NoOperandsInstruction }
type DUP2 struct{ base.NoOperandsInstruction }
type DUP2_X1 struct{ base.NoOperandsInstruction }
type DUP2_X2 struct{ base.NoOperandsInstruction }
```

dup 指令复制栈顶的单个变量，代码如下：

```
func (self *DUP) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    slot := stack.PopSlot()
```

```
    stack.PushSlot(slot)
    stack.PushSlot(slot)
}
```

其他 5 条指令和 dup 指令还是有一定差别的，这里就不具体介绍了，请读者阅读随书源代码。

5.6.3 swap 指令

在 ch05\instructions\stack 目录下创建 swap.go 文件，在其中定义 swap 指令，代码如下：

```
package stack

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Swap the top two operand stack values
type SWAP struct{ base.NoOperandsInstruction }
```

swap 指令交换栈顶的两个变量，Execute() 方法如下：

```
func (self *SWAP) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    slot1 := stack.PopSlot()
    slot2 := stack.PopSlot()
    stack.PushSlot(slot1)
    stack.PushSlot(slot2)
}
```

5.7 数学指令

数学指令大致对应 Java 语言中的加、减、乘、除等数学运算符。数学指令包括算术指令、位移指令和布尔运算指令等，共 37 条，将全部在本节实现。

5.7.1 算术指令

算术指令又可以进一步分为加法（add）指令、减法（sub）指令、乘法（mul）指令、除法（div）指令、求余（rem）指令和取反（neg）指令 6 种。加、减、乘、除和取反指令都比较简单，本节以稍微复杂一些的求余指令为例进行讨论。

在 ch05\instructions\math 目录下创建 rem.go 文件，在其中定义 4 条求余指令，代码如下：

```
package math

import "math"
import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type DREM struct{ base.NoOperandsInstruction }
type FREM struct{ base.NoOperandsInstruction }
type IREM struct{ base.NoOperandsInstruction }
type LREM struct{ base.NoOperandsInstruction }
```

irem 和 lrem 代码差不多，以 irem 为例，其 Execute() 方法如下：

```
func (self *IREM) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    v2 := stack.PopInt()
    v1 := stack.PopInt()
    if v2 == 0 {
        panic("java.lang.ArithmeticException: / by zero")
    }

    result := v1 % v2
    stack.PushInt(result)
}
```

先从操作数栈中弹出两个 int 变量，求余，然后把结果推入操作数栈。这里注意一点，对 int 或 long 变量做除法和求余运算时，是有可能抛出 ArithmeticException 异常的。frem 和 drem 指令差不多，以 drem 为例，其 Execute() 方法如下：

```
func (self *DREM) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    v2 := stack.PopDouble()
    v1 := stack.PopDouble()
    result := math.Mod(v1, v2)
    stack.PushDouble(result)
}
```

Go 语言没有给浮点数类型定义求余操作符，所以需要使用 math 包的 Mod() 函数。另外，浮点数类型因为有 Infinity（无穷大）值，所以即使是除零，也不会导致 ArithmeticException 异常抛出。

5.7.2　位移指令

位移指令可以分为左移和右移两种，右移指令又可以分为算术右移（有符号右移）和

逻辑右移（无符号右移）两种。算术右移和逻辑位移的区别仅在于符号位的扩展，如下面的 Java 代码所示。

```
int x = -1;
println(Integer.toBinaryString(x));        // 11111111111111111111111111111111
println(Integer.toBinaryString(x >> 8));   // 11111111111111111111111111111111
println(Integer.toBinaryString(x >>> 8)); // 00000000111111111111111111111111
```

在 ch05\instructions\math 目录下创建 sh.go 文件，在其中定义 6 条位移指令，代码如下：

```
package math

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type ISHL struct{ base.NoOperandsInstruction } // int 左位移
type ISHR struct{ base.NoOperandsInstruction } // int 算术右位移
type IUSHR struct{ base.NoOperandsInstruction }// int 逻辑右位移
type LSHL struct{ base.NoOperandsInstruction } // long 左位移
type LSHR struct{ base.NoOperandsInstruction } // long 算术右位移
type LUSHR struct{ base.NoOperandsInstruction }// long 逻辑右位移
```

左移指令比较简单，以 ishl 指令为例，其 Execute() 方法如下：

```
func (self *ISHL) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    v2 := stack.PopInt()
    v1 := stack.PopInt()
    s := uint32(v2) & 0x1f
    result := v1 << s
    stack.PushInt(result)
}
```

先从操作数栈中弹出两个 int 变量 v2 和 v1。v1 是要进行位移操作的变量，v2 指出要移位多少比特。位移之后，把结果推入操作数栈。这里注意两点：第一，int 变量只有 32 位，所以只取 v2 的前 5 个比特就足够表示位移位数了；第二，Go 语言位移操作符右侧必须是无符号整数，所以需要对 v2 进行类型转换。

算术右移指令需要扩展符号位，代码和左移指令基本上差不多。以 lshr 指令为例，其 Execute() 方法如下：

```
func (self *LSHR) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    v2 := stack.PopInt()
    v1 := stack.PopLong()
```

```
    s := uint32(v2) & 0x3f
    result := v1 >> s
    stack.PushLong(result)
}
```

long 变量有 64 位，所以取 v2 的前 6 个比特。最后以 iushr 为例，介绍逻辑右移指令是如何实现的。

```
func (self *IUSHR) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    v2 := stack.PopInt()
    v1 := stack.PopInt()
    s := uint32(v2) & 0x1f
    result := int32(uint32(v1) >> s)
    stack.PushInt(result)
}
```

Go 语言并没有 Java 语言中的 >>> 运算符，为了达到无符号位移的目的，需要先把 v1 转成无符号整数，位移操作之后，再转回有符号整数。

5.7.3　布尔运算指令

布尔运算指令只能操作 int 和 long 变量，分为按位与（and）、按位或（or）、按位异或（xor）3 种。以按位与为例介绍布尔运算指令。在 ch05\instructions\math 目录下创建 and.go 文件，在其中定义 iand 和 land 指令，代码如下：

```
package math

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type IAND struct{ base.NoOperandsInstruction }
type LAND struct{ base.NoOperandsInstruction }
```

以 iand 指令为例，其 Execute() 方法如下：

```
func (self *IAND) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    v2 := stack.PopInt()
    v1 := stack.PopInt()
    result := v1 & v2
    stack.PushInt(result)
}
```

代码比较简单，就不多解释了。

5.7.4 iinc 指令

iinc 指令给局部变量表中的 int 变量增加常量值，局部变量表索引和常量值都由指令的操作数提供。在 ch05\instructions\math 目录下创建 iinc.go 文件，在其中定义 iinc 指令，代码如下：

```
package math

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Increment local variable by constant
type IINC struct {
    Index uint
    Const int32
}
```

FetchOperands() 函数从字节码里读取操作数，代码如下：

```
func (self *IINC) FetchOperands(reader *base.BytecodeReader) {
    self.Index = uint(reader.ReadUint8())
    self.Const = int32(reader.ReadInt8())
}
```

Execute() 方法从局部变量表中读取变量，给它加上常量值，再把结果写回局部变量表，代码如下：

```
func (self *IINC) Execute(frame *rtda.Frame) {
    localVars := frame.LocalVars()
    val := localVars.GetInt(self.Index)
    val += self.Const
    localVars.SetInt(self.Index, val)
}
```

5.8 类型转换指令

类型转换指令大致对应 Java 语言中的基本类型强制转换操作。类型转换指令有共 15 条，将全部在本节实现。引用类型转换对应的是 checkcast 指令，将在第 6 章介绍。

按照被转换变量的类型，类型转换指令可以分为 3 种：i2x 系列指令把 int 变量强制转换成其他类型；12x 系列指令把 long 变量强制转换成其他类型；f2x 系列指令把 float 变量强制转换成其他类型；d2x 系列指令把 double 变量强制转换成其他类型。以 d2x 系列指令

为例进行讨论。

在 ch05\instructions\ conversions 目录下创建 d2x.go 文件，在其中定义 d2f、d2i 和 d2l 指令，代码如下：

```
package conversions

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type D2F struct{ base.NoOperandsInstruction }
type D2I struct{ base.NoOperandsInstruction }
type D2L struct{ base.NoOperandsInstruction }
```

以 d2i 指令为例，它的 Execute() 方法如下：

```
func (self *D2I) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    d := stack.PopDouble()
    i := int32(d)
    stack.PushInt(i)
}
```

因为 Go 语言可以很方便地转换各种基本类型的变量，所以类型转换指令实现起来还是比较容易的。

5.9　比较指令

比较指令可以分为两类：一类将比较结果推入操作数栈顶，一类根据比较结果跳转。比较指令是编译器实现 if-else、for、while 等语句的基石，共有 19 条，将全部在本节实现。

5.9.1　lcmp 指令

lcmp 指令用于比较 long 变量。在 ch05\instructions\comparisons 目录下创建 lcmp.go 文件，在其中定义 lcmp 指令，代码如下：

```
package comparisons

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Compare long
```

```
type LCMP struct{ base.NoOperandsInstruction }
```

Execute() 方法把栈顶的两个 long 变量弹出，进行比较，然后把比较结果（int 型 0、1 或 -1）推入栈顶，代码如下：

```
func (self *LCMP) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    v2 := stack.PopLong()
    v1 := stack.PopLong()
    if v1 > v2 {
        stack.PushInt(1)
    } else if v1 == v2 {
        stack.PushInt(0)
    } else {
        stack.PushInt(-1)
    }
}
```

5.9.2 fcmp<op> 和 dcmp<op> 指令

fcmpg 和 fcmpl 指令用于比较 float 变量。在 ch05\instructions\comparisons 目录下创建 fcmp.go 文件，在其中定义 fcmpg 和 fcmpl 指令，代码如下：

```
package comparisons

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Compare float
type FCMPG struct{ base.NoOperandsInstruction }
type FCMPL struct{ base.NoOperandsInstruction }
```

这两条指令和 lcmp 指令很像，但是除了比较的变量类型不同以外，还有一个重要的区别。由于浮点数计算有可能产生 NaN（Not a Number）值，所以比较两个浮点数时，除了大于、等于、小于之外，还有第 4 种结果：无法比较。fcmpg 和 fcmpl 指令的区别就在于对第 4 种结果的定义。编写一个函数来统一比较 float 变量，代码如下：

```
func _fcmp(frame *rtda.Frame, gFlag bool) {
    stack := frame.OperandStack()
    v2 := stack.PopFloat()
    v1 := stack.PopFloat()
    if v1 > v2 {
        stack.PushInt(1)
    } else if v1 == v2 {
```

```
        stack.PushInt(0)
    } else if v1 < v2 {
        stack.PushInt(-1)
    } else if gFlag {
        stack.PushInt(1)
    } else {
        stack.PushInt(-1)
    }
}
```

fcmpg 和 fcmpl 指令的 Execute() 方法只是简单地调用 _fcmp() 函数而已，代码如下：

```
func (self *FCMPG) Execute(frame *rtda.Frame) {
    _fcmp(frame, true)
}
func (self *FCMPL) Execute(frame *rtda.Frame) {
    _fcmp(frame, false)
}
```

也就是说，当两个 float 变量中至少有一个是 NaN 时，用 fcmpg 指令比较的结果是 1，而用 fcmpl 指令比较的结果是 -1。

dcmpg 和 dcmpl 指令用来比较 double 变量，在 dcmp.go 文件中，这两条指令和 fcmpg、fcmpl 指令除了比较的变量类型不同之外，代码基本上完全一样，这里就不详细介绍了。

5.9.3 if<cond> 指令

在 ch05\instructions\comparisons 目录下创建 ifcond.go 文件，在其中定义 6 条 if<cond> 指令，代码如下：

```
package comparisons

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Branch if int comparison with zero succeeds
type IFEQ struct{ base.BranchInstruction }
type IFNE struct{ base.BranchInstruction }
type IFLT struct{ base.BranchInstruction }
type IFLE struct{ base.BranchInstruction }
type IFGT struct{ base.BranchInstruction }
type IFGE struct{ base.BranchInstruction }
```

if<cond> 指令把操作数栈顶的 int 变量弹出，然后跟 0 进行比较，满足条件则跳转。

假设从栈顶弹出的变量是 x，则指令执行跳转操作的条件如下：

- ifeq：x == 0
- ifne：x != 0
- iflt：x < 0
- ifle：x <= 0
- ifgt：x > 0
- ifge：x >= 0

以 ifeq 指令为例，其 Execute() 方法如下：

```
func (self *IFEQ) Execute(frame *rtda.Frame) {
    val := frame.OperandStack().PopInt()
    if val == 0 {
        base.Branch(frame, self.Offset)
    }
}
```

真正的跳转逻辑在 Branch() 函数中。因为这个函数在很多指令中都会用到，所以把它定义在 ch05\instructions\base\branch_logic.go 文件中，代码如下：

```
package base

import "jvmgo/ch05/rtda"

func Branch(frame *rtda.Frame, offset int) {
    pc := frame.Thread().PC()
    nextPC := pc + offset
    frame.SetNextPC(nextPC)
}
```

Frame 结构体的 SetNextPC() 方法将在 5.12 小节介绍。

5.9.4 if_icmp<cond> 指令

在 ch05\instructions\comparisons 目录下创建 if_icmp.go 文件，在其中定义 6 条 if_ icmp 指令，代码如下：

```
package comparisons

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"
```

```
// Branch if int comparison succeeds
type IF_ICMPEQ struct{ base.BranchInstruction }
type IF_ICMPNE struct{ base.BranchInstruction }
type IF_ICMPLT struct{ base.BranchInstruction }
type IF_ICMPLE struct{ base.BranchInstruction }
type IF_ICMPGT struct{ base.BranchInstruction }
type IF_ICMPGE struct{ base.BranchInstruction }
```

if_icmp<cond> 指令把栈顶的两个 int 变量弹出，然后进行比较，满足条件则跳转。跳转条件和 if<cond> 指令类似。以 if_icmpne 指令为例，其 Execute() 方法如下：

```
func (self *IF_ICMPNE) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    val2 := stack.PopInt()
    val1 := stack.PopInt()
    if val1 != val2 {
        base.Branch(frame, self.Offset)
    }
}
```

5.9.5　if_acmp<cond> 指令

在 ch05\instructions\comparisons 目录下创建 if_acmp.go 文件，在其中定义两条 if_ acmp <cond> 指令，代码如下：

```
package comparisons

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Branch if reference comparison succeeds
type IF_ACMPEQ struct{ base.BranchInstruction }
type IF_ACMPNE struct{ base.BranchInstruction }
```

if_acmpeq 和 if_acmpne 指令把栈顶的两个引用弹出，根据引用是否相同进行跳转。以 if_acmpeq 指令为例，其 Execute() 方法如下：

```
func (self *IF_ACMPEQ) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    ref2 := stack.PopRef()
    ref1 := stack.PopRef()
    if ref1 == ref2 {
        base.Branch(frame, self.Offset)
    }
}
```

5.10 控制指令

控制指令共有 11 条。jsr 和 ret 指令在 Java 6 之前用于实现 finally 子句，从 Java 6 开始，Oracle 的 Java 编译器已经不再使用这两条指令了，本书不讨论这两条指令。return 系列指令有 6 条，用于从方法调用中返回，将在第 7 章讨论方法调用和返回时实现这 6 条指令。本节实现剩下的 3 条指令：goto、tableswitch 和 lookupswitch。

5.10.1 goto 指令

在 ch05\instructions\control 目录下创建 goto.go 文件，在其中定义 goto 指令，代码如下：

```
package control

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Branch always
type GOTO struct{ base.BranchInstruction }
```

goto 指令进行无条件跳转，其 Execute() 方法如下：

```
func (self *GOTO) Execute(frame *rtda.Frame) {
   base.Branch(frame, self.Offset)
}
```

5.10.2 tableswitch 指令

Java 语言中的 switch-case 语句有两种实现方式：如果 case 值可以编码成一个索引表，则实现成 tableswitch 指令；否则实现成 lookupswitch 指令。Java 虚拟机规范的 3.10 小节里有两个例子，我们可以借用一下。下面这个 Java 方法中的 switch-case 可以编译成 tableswitch 指令，代码如下：

```
int chooseNear(int i) {
   switch (i) {
      case 0:  return  0;
      case 1:  return  1;
      case 2:  return  2;
      default: return -1;
   }
}
```

下面这个 Java 方法中的 switch-case 则需要编译成 lookupswitch 指令：

```
int chooseFar(int i) {
    switch (i) {
        case -100: return -1;
        case 0:    return  0;
        case 100:  return  1;
        default:   return -1;
    }
}
```

在 ch05\instructions\control 目录下创建 tableswitch.go 文件，在其中定义 tableswitch 指令，代码如下：

```
package control

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

// Access jump table by index and jump
type TABLE_SWITCH struct {
    defaultOffset   int32
    low             int32
    high            int32
    jumpOffsets     []int32
}
```

tableswitch 指令的操作数比较复杂，它的 FetchOperands() 方法如下：

```
func (self *TABLE_SWITCH) FetchOperands(reader *base.BytecodeReader) {
    reader.SkipPadding()
    self.defaultOffset = reader.ReadInt32()
    self.low = reader.ReadInt32()
    self.high = reader.ReadInt32()
    jumpOffsetsCount := self.high - self.low + 1
    self.jumpOffsets = reader.ReadInt32s(jumpOffsetsCount)
}
```

tableswitch 指令操作码的后面有 0 ~ 3 字节的 padding，以保证 defaultOffset 在字节码中的地址是 4 的倍数。BytecodeReader 结构体的 SkipPadding() 方法如下：

```
func (self *BytecodeReader) SkipPadding() {
    for self.pc%4 != 0 {
        self.ReadUint8()
    }
}
```

defaultOffset 对应默认情况下执行跳转所需的字节码偏移量；low 和 high 记录 case 的

取值范围；jumpOffsets 是一个索引表，里面存放 high-low+1 个 int 值，对应各种 case 情况下，执行跳转所需的字节码偏移量。BytecodeReader 结构体的 ReadInt32s() 方法如下：

```
func (self *BytecodeReader) ReadInt32s(n int32) []int32 {
    ints := make([]int32, n)
    for i := range ints {
        ints[i] = self.ReadInt32()
    }
    return ints
}
```

Execute() 方法先从操作数栈中弹出一个 int 变量，然后看它是否在 low 和 high 给定的范围之内。如果在，则从 jumpOffsets 表中查出偏移量进行跳转，否则按照 defaultOffset 跳转。代码如下：

```
func (self *TABLE_SWITCH) Execute(frame *rtda.Frame) {
    index := frame.OperandStack().PopInt()

    var offset int
    if index >= self.low && index <= self.high {
        offset = int(self.jumpOffsets[index-self.low])
    } else {
        offset = int(self.defaultOffset)
    }

    base.Branch(frame, offset)
}
```

5.10.3 lookupswitch 指令

在 ch05\instructions\control 目录下创建 lookupswitch.go 文件，在其中定义 lookupswitch 指令，代码如下：

```
package control

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type LOOKUP_SWITCH struct {
    defaultOffset   int32
    npairs          int32
    matchOffsets    []int32
}
```

FetchOperands() 方法也要先跳过 padding，代码如下：

```
func (self *LOOKUP_SWITCH) FetchOperands(reader *base.BytecodeReader) {
    reader.SkipPadding()
    self.defaultOffset = reader.ReadInt32()
    self.npairs = reader.ReadInt32()
    self.matchOffsets = reader.ReadInt32s(self.npairs * 2)
}
```

matchOffsets 有点像 Map，它的 key 是 case 值，value 是跳转偏移量。Execute() 方法先从操作数栈中弹出一个 int 变量，然后用它查找 matchOffsets，看是否能找到匹配的 key。如果能，则按照 value 给出的偏移量跳转，否则按照 defaultOffset 跳转。代码如下：

```
func (self *LOOKUP_SWITCH) Execute(frame *rtda.Frame) {
    key := frame.OperandStack().PopInt()
    for i := int32(0); i < self.npairs*2; i += 2 {
        if self.matchOffsets[i] == key {
            offset := self.matchOffsets[i+1]
            base.Branch(frame, int(offset))
            return
        }
    }
    base.Branch(frame, int(self.defaultOffset))
}
```

5.11　扩展指令

扩展指令共有 6 条。和 jsr 指令一样，本书不讨论 jsr_w 指令。multianewarray 指令用于创建多维数组，在第 8 章讨论数组时实现该指令。本节实现剩下的 4 条指令。

5.11.1　wide 指令

加载类指令、存储类指令、ret 指令和 iinc 指令需要按索引访问局部变量表，索引以 uint8 的形式存在字节码中。对于大部分方法来说，局部变量表大小都不会超过 256，所以用一字节来表示索引就够了。但是如果有方法的局部变量表超过这限制呢？ Java 虚拟机规范定义了 wide 指令来扩展前述指令。

在 ch05\instructions\extended 目录下创建 wide.go 文件，在其中定义 wide 指令，代码如下：

```
package extended
```

```
import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/instructions/loads"
import "jvmgo/ch05/instructions/math"
import "jvmgo/ch05/instructions/stores"
import "jvmgo/ch05/rtda"

// Extend local variable index by additional bytes
type WIDE struct {
    modifiedInstruction base.Instruction
}
```

wide 指令改变其他指令的行为，modifiedInstruction 字段存放被改变的指令。wide 指令需要自己解码出 modifiedInstruction，FetchOperands() 方法的代码如下：

```
func (self *WIDE) FetchOperands(reader *base.BytecodeReader) {
    opcode := reader.ReadUint8()
    switch opcode {
    case 0x15: ... // iload
    case 0x16: ... // lload
    case 0x17: ... // fload
    case 0x18: ... // dload
    case 0x19: ... // aload
    case 0x36: ... // istore
    case 0x37: ... // lstore
    case 0x38: ... // fstore
    case 0x39: ... // dstore
    case 0x3a: ... // astore
    case 0x84: ... // iinc
    case 0xa9: // ret
        panic("Unsupported opcode: 0xa9!")
    }
}
```

FetchOperands() 方法先从字节码中读取一字节的操作码，然后创建子指令实例，最后读取子指令的操作数。因为没有实现 ret 指令，所以暂时调用 panic() 函数终止程序执行。加载指令和存储指令都只有一个操作数，需要扩展成 2 字节，以 iload 为例：

```
case 0x15:
    inst := &loads.ILOAD{}
    inst.Index = uint(reader.ReadUint16())
    self.modifiedInstruction = inst
```

iinc 指令有两个操作数，都需要扩展成 2 字节，代码如下：

```
case 0x84:
    inst := &math.IINC{}
```

```
    inst.Index = uint(reader.ReadUint16())
    inst.Const = int32(reader.ReadInt16())
    self.modifiedInstruction = inst
```

wide 指令只是增加了索引宽度，并不改变子指令操作，所以其 Execute() 方法只要调用子指令的 Execute() 方法即可，代码如下：

```
func (self *WIDE) Execute(frame *rtda.Frame) {
    self.modifiedInstruction.Execute(frame)
}
```

5.11.2 ifnull 和 ifnonnull 指令

在 ch05\instructions\extended 目录下创建 ifnull.go 文件，在其中定义 ifnull 和 ifnonnull 指令，代码如下：

```
package extended

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

type IFNULL struct{ base.BranchInstruction } // Branch if reference is null
type IFNONNULL struct{ base.BranchInstruction } // Branch if reference not null
```

根据引用是否是 null 进行跳转，ifnull 和 ifnonnull 指令把栈顶的引用弹出。以 ifnull 指令为例，它的 Execute() 方法如下：

```
func (self *IFNULL) Execute(frame *rtda.Frame) {
    ref := frame.OperandStack().PopRef()
    if ref == nil {
        base.Branch(frame, self.Offset)
    }
}
```

5.11.3 goto_w 指令

在 ch05\instructions\extended 目录下创建 goto_w.go 文件，在其中定义 goto_w 指令，代码如下：

```
package extended

import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"
```

```
// Branch always (wide index)
type GOTO_W struct {
    offset int
}
```

goto_w 指令和 goto 指令的唯一区别就是索引从 2 字节变成了 4 字节。FetchOperands() 方法代码如下：

```
func (self *GOTO_W) FetchOperands(reader *base.BytecodeReader) {
    self.offset = int(reader.ReadInt32())
}
```

Execute() 方法的代码如下：

```
func (self *GOTO_W) Execute(frame *rtda.Frame) {
    base.Branch(frame, self.offset)
}
```

5.12 解释器

指令集已经实现得差不多了，本节编写一个简单的解释器。这个解释器目前只能执行一个 Java 方法，但是在后面的章节中，会不断完善它，让它变得越来越强大。在 ch05 目录下创建 interpreter.go 文件，在其中定义 interpret() 函数，代码如下：

```
package main

import "fmt"
import "jvmgo/ch05/classfile"
import "jvmgo/ch05/instructions"
import "jvmgo/ch05/instructions/base"
import "jvmgo/ch05/rtda"

func interpret(methodInfo *classfile.MemberInfo) {...}
```

interpret() 方法的参数是 MemberInfo 指针，调用 MemberInfo 结构体的 CodeAttribute() 方法可以获取它的 Code 属性，语法结构如下：

```
func interpret(methodInfo *classfile.MemberInfo) {
    codeAttr := methodInfo.CodeAttribute()
    ... // 其他代码
}
```

CodeAttribute() 方法是新增加的，代码在 ch05\classfile\member_info.go 文件中，代码

如下：

```
func (self *MemberInfo) CodeAttribute() *CodeAttribute {
    for _, attrInfo := range self.attributes {
        switch attrInfo.(type) {
        case *CodeAttribute:
            return attrInfo.(*CodeAttribute)
        }
    }
    return nil
}
```

得到Code属性之后，可以进一步获得执行方法所需的局部变量表和操作数栈空间，以及方法的字节码。

```
func interpret(methodInfo *classfile.MemberInfo) {
    codeAttr := methodInfo.CodeAttribute()
    maxLocals := codeAttr.MaxLocals()
    maxStack := codeAttr.MaxStack()
    bytecode := codeAttr.Code()
    ... // 其他代码
}
```

interpret()方法的其余代码先创建一个Thread实例，然后创建一个帧并把它推入Java虚拟机栈顶，最后执行方法。完整的代码如下：

```
func interpret(methodInfo *classfile.MemberInfo) {
    codeAttr := methodInfo.CodeAttribute()
    maxLocals := codeAttr.MaxLocals()
    maxStack := codeAttr.MaxStack()
    bytecode := codeAttr.Code()

    thread := rtda.NewThread()
    frame := thread.NewFrame(maxLocals, maxStack)
    thread.PushFrame(frame)

    defer catchErr(frame)
    loop(thread, bytecode)
}
```

Thread结构体的NewFrame()方法是新增加的，代码在ch05\rtda\thread.go文件中，如下所示：

```
func (self *Thread) NewFrame(maxLocals, maxStack uint) *Frame {
    return newFrame(self, maxLocals, maxStack)
}
```

Frame 结构体也有变化，增加了两个字段，改动如下（在 ch05\rtda\frame.go 文件中）：

```
type Frame struct {
    lower          *Frame
    localVars      LocalVars
    operandStack   *OperandStack
    thread         *Thread
    nextPC         int
}
```

这两个字段主要是为了实现跳转指令而添加的，回顾一下 Branch() 方法，代码如下：

```
func Branch(frame *rtda.Frame, offset int) {
    pc := frame.Thread().PC()
    nextPC := pc + offset
    frame.SetNextPC(nextPC)
}
```

Frame 结构体的 newFrame() 方法也相应发生了变化，改动如下：

```
func newFrame(thread *Thread, maxLocals, maxStack uint) *Frame {
    return &Frame{
        thread:       thread,
        localVars:    newLocalVars(maxLocals),
        operandStack: newOperandStack(maxStack),
    }
}
```

回到 interpret() 方法，我们的解释器目前还没有办法优雅地结束运行。因为每个方法的最后一条指令都是某个 return 指令，而还没有实现 return 指令，所以方法在执行过程中必定会出现错误，此时解释器逻辑会转到 catchErr() 函数，代码如下：

```
func catchErr(frame *rtda.Frame) {
    if r := recover(); r != nil {
        fmt.Printf("LocalVars:%v\n", frame.LocalVars())
        fmt.Printf("OperandStack:%v\n", frame.OperandStack())
        panic(r)
    }
}
```

把局部变量表和操作数栈的内容打印出来，以此来观察方法的执行结果。还剩一个 loop() 函数，其代码如下：

```
func loop(thread *rtda.Thread, bytecode []byte) {
    frame := thread.PopFrame()
    reader := &base.BytecodeReader{}
```

```
        for {
            pc := frame.NextPC()
            thread.SetPC(pc)

            // decode
            reader.Reset(bytecode, pc)
            opcode := reader.ReadUint8()
            inst := instructions.NewInstruction(opcode)
            inst.FetchOperands(reader)
            frame.SetNextPC(reader.PC())

            // execute
            fmt.Printf("pc:%2d inst:%T %v\n", pc, inst, inst)
            inst.Execute(frame)
        }
    }
```

loop() 函数循环执行“计算 pc、解码指令、执行指令”这三个步骤，直到遇到错误！代码中有一个函数还没有给出代码：NewInstruction()。这个函数是 switch-case 语句，根据操作码创建具体的指令，代码在 ch05\instructions\factory.go 文件中，如下所示：

```
package instructions

import "fmt"
import "jvmgo/ch05/instructions/base"
import . "jvmgo/ch05/instructions/comparisons"
import . "jvmgo/ch05/instructions/constants"
import . "jvmgo/ch05/instructions/control"
import . "jvmgo/ch05/instructions/conversions"
import . "jvmgo/ch05/instructions/extended"
import . "jvmgo/ch05/instructions/loads"
import . "jvmgo/ch05/instructions/math"
import . "jvmgo/ch05/instructions/stack"
import . "jvmgo/ch05/instructions/stores"

func NewInstruction(opcode byte) base.Instruction {
    switch opcode {
    case 0x00: return &NOP{}
    case 0x01: return &ACONST_NULL{}
    ...
    default:
        panic(fmt.Errorf("Unsupported opcode: 0x%x!", opcode))
    }
}
```

这个 switch-case 语句非常长，为了节约篇幅，这里就不给出全部代码了。另外，有

很大一部分指令是没有操作数的，没有必要每次都创建不同的实例。为了优化，可以给这些指令定义单例变量，代码如下：

```
var (
   nop             = &NOP{}
   aconst_null     = &ACONST_NULL{}
   ...
)
```

对于这类指令，在 NewInstruction() 函数中直接返回单例变量即可，代码如下：

```
func NewInstruction(opcode byte) base.Instruction {
   switch opcode {
   case 0x00: return nop
   case 0x01: return aconst_null
   ...
   }
}
```

5.13 测试本章代码

德国大数学家高斯有一个广为流传的小故事。话说高斯 7 岁开始上学，10 岁开始学习数学。有一天，数学老师布置了一道题：问 1+2+3……这样从 1 一直加到 100 等于多少。老师原以为孩子们要算上一段时间，可是没想到小高斯很快就给出了答案。高斯当然不是从 1 一直加到 100，而是用更聪明的办法计算的：1+100=101，2+99=101……1 加到 100 有 50 组这样的数，所以 50*101=5050。

本节用最笨的办法来计算这个题目，考验一下虚拟机是否可以工作。随书 Java 代码里有一个例子，代码如下：

```
package jvmgo.book.ch05;

public class GaussTest {
   public static void main(String[] args) {
      int sum = 0;
      for (int i = 1; i <= 100; i++) {
         sum += i;
      }
      System.out.println(sum);
   }
}
```

下面改造 ch05\main.go 文件。首先修改 import 语句，代码如下：

```
package main

import "fmt"
import "strings"
import "jvmgo/ch05/classfile"
import "jvmgo/ch05/classpath"
```

main() 函数不变，修改 startJVM() 函数，改动如下：

```
func startJVM(cmd *Cmd) {
    cp := classpath.Parse(cmd.XjreOption, cmd.cpOption)
    className := strings.Replace(cmd.class, ".", "/", -1)
    cf := loadClass(className, cp)
    mainMethod := getMainMethod(cf)
    if mainMethod != nil {
        interpret(mainMethod)
    } else {
        fmt.Printf("Main method not found in class %s\n", cmd.class)
    }
}
```

startJVM() 首先调用 loadClass() 方法读取并解析 class 文件，然后调用 getMainMethod() 函数查找类的 main() 方法，最后调用 interpret() 函数解释执行 main() 方法。loadClass() 函数的代码如下：

```
func loadClass(className string, cp *classpath.Classpath) *classfile.ClassFile {
    classData, _, err := cp.ReadClass(className)
    if err != nil {
        panic(err)
    }
    cf, err := classfile.Parse(classData)
    if err != nil {
        panic(err)
    }
    return cf
}
```

getMainMethod() 函数的代码如下：

```
func getMainMethod(cf *classfile.ClassFile) *classfile.MemberInfo {
    for _, m := range cf.Methods() {
        if m.Name() == "main" && m.Descriptor() == "([Ljava/lang/String;)V" {
            return m
        }
    }
    return nil
}
```

打开命令行窗口，执行下面的命令编译本章代码。

```
go install jvmgo\ch05
```

编译成功后，在 D:\go\workspace\bin 目录下会出现 ch05.exe 文件。用 javac 编译 GaussTest 类，然后用 ch05.exe 执行 GaussTest 程序，结果如图 5-3 所示。注意一定要保证可以在当前目录下找到 GaussTest.class 文件，否则应该用 -cp 选项指定用户类路径。

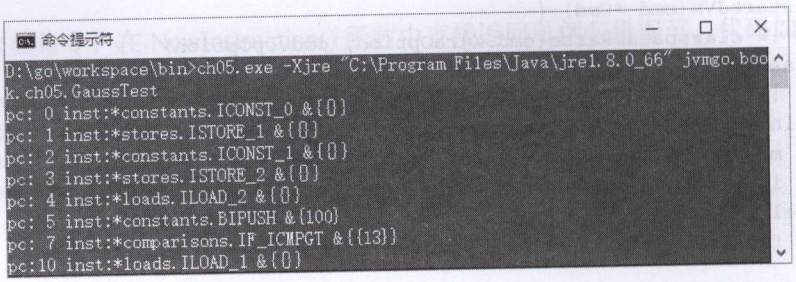


图 5-3 GaussTest 程序执行结果（1）

方法执行，并打印出了执行过的指令。在我们预料之中，方法执行的最后出现了错误，局部变量表和操作数栈的状态也打印了出来，如图 5-4 所示。仔细观察局部变量表可以看到 5050 这个数字，这正是我们的计算结果！

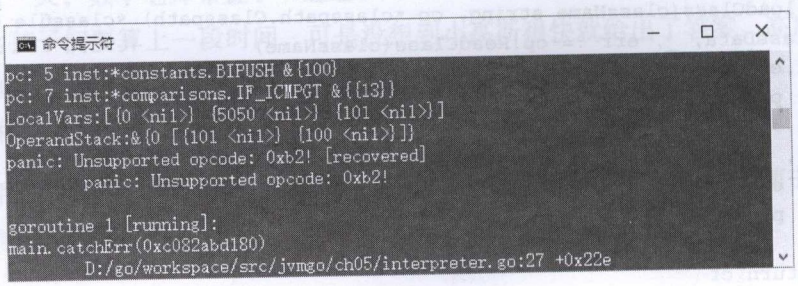
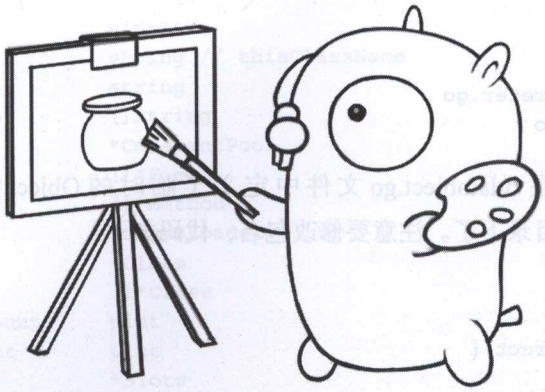


图 5-4 GaussTest 程序执行结果（2）

5.14 本章小结

虽然还有很多缺陷，但是我们的 Java 虚拟机已经可以解释执行字节码了，这是一个很大的进步！下一章将讨论类和对象在内存中的布局，并且开始实现引用类指令。还等什么？快来阅读吧！



第6章 类和对象

在第4章，我们初步实现了线程私有的运行时数据区，在此基础上，第5章实现了一个简单的解释器和150多条指令。这些指令主要是操作局部变量表和操作数栈、进行数学运算、比较运算和跳转控制等。本章将实现线程共享的运行时数据区，包括方法区和运行时常量池。

第2章实现了类路径，可以找到class文件，并把数据加载到内存中。第3章实现了class文件解析，可以把class数据解析成一个ClassFile结构体。本章将进一步处理ClassFile结构体，把它加以转换，放进方法区以供后续使用。本章还会初步讨论类和对象的设计，实现一个简单的类加载器，并且实现类和对象相关的部分指令。

在开始学习本章之前，还是先把目录结构准备好。复制ch05目录，改名为ch06。修改main.go等文件，把import语句中的ch05全都改成ch06，然后在ch06\rtda目录中创建heap子目录。现在目录结构看起来应该是下面这样：

```
D:\go\workspace\src
  |-jvmgo
    |-ch01 ~ ch05
    |-ch06
      |-classfile
      |-classpath
      |-instructions
```

```
    |-rtda
      |-heap
    |-cmd.go
    |-interpreter.go
    |-main.go
```

在第 4 章中，在 rtda\object.go 文件中定义了临时的 Object 结构体。现在可以把 object.go 移到 heap 目录下了。注意要修改包名，代码如下：

```
package heap

type Object struct {
    // todo
}
```

还需要修改 slot.go、local_vars.go 和 operand_stack.go 这三个文件，在其中添加 heap 包的 import 语句，并把 *Object 改成 *heap.Object。以上改动不大，为了节约篇幅，这里就不给出具体代码了。

6.1 方法区

第 4 章简单讨论过方法区，它是运行时数据区的一块逻辑区域，由多个线程共享。方法区主要存放从 class 文件获取的类信息。此外，类变量也存放在方法区中。当 Java 虚拟机第一次使用某个类时，它会搜索类路径，找到相应的 class 文件，然后读取并解析 class 文件，把相关信息放进方法区。至于方法区到底位于何处，是固定大小还是动态调整，是否参与垃圾回收，以及如何在方法区内存放类数据等，Java 虚拟机规范并没有明确规定。

先来看看有哪些信息需要放进方法区。

6.1.1 类信息

使用结构体来表示将要放进方法区内的类。在 ch06\rtda\heap 目录下创建 class.go 文件，在其中定义 Class 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"
```

```
type Class struct {
    accessFlags             uint16
    name                    string // thisClassName
    superClassName          string
    interfaceNames          []string
    constantPool            *ConstantPool
    fields                  []*Field
    methods                 []*Method
    loader                  *ClassLoader
    superClass              *Class
    interfaces              []*Class
    instanceSlotCount       uint
    staticSlotCount         uint
    staticVars              *Slots
}
```

accessFlags 是类的访问标志，总共 16 比特。字段和方法也有访问标志，但具体标志位的含义可能有所不同。根据 Java 虚拟机规范，把各个比特位的含义统一定义在 heap\access_flags.go 文件中，代码如下：

```
package heap

const (
    ACC_PUBLIC              = 0x0001 // class field method
    ACC_PRIVATE             = 0x0002 //       field method
    ACC_PROTECTED           = 0x0004 //       field method
    ACC_STATIC              = 0x0008 //       field method
    ACC_FINAL               = 0x0010 // class field method
    ACC_SUPER               = 0x0020 // class
    ACC_SYNCHRONIZED        = 0x0020 //             method
    ACC_VOLATILE            = 0x0040 //       field
    ACC_BRIDGE              = 0x0040 //             method
    ACC_TRANSIENT           = 0x0080 //       field
    ACC_VARARGS             = 0x0080 //             method
    ACC_NATIVE              = 0x0100 //             method
    ACC_INTERFACE           = 0x0200 // class
    ACC_ABSTRACT            = 0x0400 // class       method
    ACC_STRICT              = 0x0800 //             method
    ACC_SYNTHETIC           = 0x1000 // class field method
    ACC_ANNOTATION          = 0x2000 // class
    ACC_ENUM                = 0x4000 // class field
)
```

回到 Class 结构体。name、superClassName 和 interfaceNames 字段分别存放类名、超类名和接口名。注意这些类名都是完全限定名，具有 java/lang/Object 的形式。constantPool 字段存放运行时常量池指针，fields 和 methods 字段分别存放字段表和方法表。运行时常

量池将在 6.2 节中详细介绍。

继续编辑 class.go 文件，在其中定义 newClass() 函数，用来把 ClassFile 结构体转换成 Class 结构体，代码如下：

```
func newClass(cf *classfile.ClassFile) *Class {
    class := &Class{}
    class.accessFlags = cf.AccessFlags()
    class.name = cf.ClassName()
    class.superClassName = cf.SuperClassName()
    class.interfaceNames = cf.InterfaceNames()
    class.constantPool = newConstantPool(class, cf.ConstantPool()) // 见 6.2 小节
    class.fields = newFields(class, cf.Fields()) // 见 6.1.2 小节
    class.methods = newMethods(class, cf.Methods()) // 见 6.1.3 小节
    return class
}
```

newClass() 函数又调用了 newConstantPool()、newFields() 和 newMethods()，这三个函数的代码将在后面的小节给出。继续编辑 class.go 文件，在其中定义 8 个方法，用来判断某个访问标志是否被设置。这 8 个方法都很简单，为了节约篇幅，这里只给出 IsPublic() 方法的代码。

```
func (self *Class) IsPublic() bool {
    return 0 != self.accessFlags&ACC_PUBLIC
}
```

后面将要介绍的 Field 和 Method 结构体也有类似的方法，届时也将不再赘述，请读者注意。

6.1.2 字段信息

字段和方法都属于类的成员，它们有一些相同的信息（访问标志、名字、描述符）。为了避免重复代码，创建一个结构体存放这些信息。在 ch06\rtda\heap 目录下创建 class_member.go 文件，在其中定义 ClassMember 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type ClassMember struct {
    accessFlags     uint16
    name            string
    descriptor      string
```

```
    class           *Class
}

func (self *ClassMember) copyMemberInfo(memberInfo *classfile.MemberInfo) {...}
```

前面三个字段的含义很明显，这里不多解释。class 字段存放 Class 结构体指针，这样可以通过字段或方法访问到它所属的类。copyMemberInfo() 方法从 class 文件中复制数据，代码如下：

```
func (self *ClassMember) copyMemberInfo(memberInfo *classfile.MemberInfo) {
    self.accessFlags = memberInfo.AccessFlags()
    self.name = memberInfo.Name()
    self.descriptor = memberInfo.Descriptor()
}
```

ClassMember 定义好了，接下来在 ch06\rtda\heap 目录下创建 field.go 文件，在其中定义 Field 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type Field struct {
    ClassMember
}

func newFields(class *Class, cfFields []*classfile.MemberInfo) []*Field {...}
```

Field 结构体比较简单，目前所有信息都是从 ClassMember 中继承过来的。newFields() 函数根据 class 文件的字段信息创建字段表，代码如下：

```
func newFields(class *Class, cfFields []*classfile.MemberInfo) []*Field {
    fields := make([]*Field, len(cfFields))
    for i, cfField := range cfFields {
        fields[i] = &Field{}
        fields[i].class = class
        fields[i].copyMemberInfo(cfField)
    }
    return fields
}
```

6.1.3 方法信息

方法比字段稍微复杂一些，因为方法中有字节码。在 ch06\rtda\heap 目录下创建

method.go 文件，在其中定义 Method 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type Method struct {
    ClassMember
    maxStack        uint
    maxLocals       uint
    code            []byte
}

func newMethods(class *Class, cfMethods []*classfile.MemberInfo) []*Method {...}
```

maxStack 和 maxLocals 字段分别存放操作数栈和局部变量表大小，这两个值是由 Java 编译器计算好的。code 字段存放方法字节码。newMethods() 函数根据 class 文件中的方法信息创建 Method 表，代码如下：

```
func newMethods(class *Class, cfMethods []*classfile.MemberInfo) []*Method {
    methods := make([]*Method, len(cfMethods))
    for i, cfMethod := range cfMethods {
        methods[i] = &Method{}
        methods[i].class = class
        methods[i].copyMemberInfo(cfMethod)
        methods[i].copyAttributes(cfMethod)
    }
    return methods
}
```

大家还记得吗？ maxStack、maxLocals 和字节码在 class 文件中是以属性的形式存储在 method_info 结构中的。如果读者已经忘记的话，可以参考 3.4.5 小节。copyAttributes() 方法从 method_info 结构中提取这些信息，代码如下：

```
func (self *Method) copyAttributes(cfMethod *classfile.MemberInfo) {
    if codeAttr := cfMethod.CodeAttribute(); codeAttr != nil {
        self.maxStack = codeAttr.MaxStack()
        self.maxLocals = codeAttr.MaxLocals()
        self.code = codeAttr.Code()
    }
}
```

到此为止，除了 ConstantPool 还没有介绍以外，已经定义了 4 个结构体，这些结构体之间的关系如图 6-1 所示。

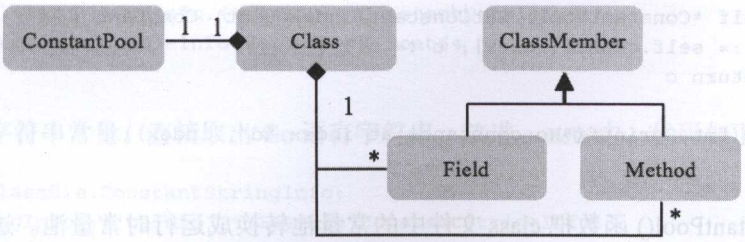


图 6-1 Class 结构体关系图

6.1.4 其他信息

Class 结构体还有几个字段没有说明。loader 字段存放类加载器指针，superClass 和 interfaces 字段存放类的超类和接口指针，这三个字段将在 6.3 节介绍。staticSlotCount 和 instanceSlotCount 字段分别存放类变量和实例变量占据的空间大小，staticVars 字段存放静态变量，这三个字段将在 6.4 节介绍。

6.2 运行时常量池

运行时常量池主要存放两类信息：字面量（literal）和符号引用（symbolic reference）。字面量包括整数、浮点数和字符串字面量；符号引用包括类符号引用、字段符号引用、方法符号引用和接口方法符号引用。在 ch06\rtda\heap 目录下创建 constant_pool.go 文件，在其中定义 Constant 接口和 ConstantPool 结构体，代码如下：

```
package heap

import "fmt"
import "jvmgo/ch06/classfile"

type Constant interface{}

type ConstantPool struct {
    class  *Class
    consts []Constant
}

func newConstantPool(class *Class, cfCp classfile.ConstantPool) *ConstantPool {...}
func (self *ConstantPool) GetConstant(index uint) Constant {...}
```

GetConstant() 方法根据索引返回常量，代码如下：

```
func (self *ConstantPool) GetConstant(index uint) Constant {
    if c := self.consts[index]; c != nil {
        return c
    }
    panic(fmt.Sprintf("No constants at index %d", index))
}
```

newConstantPool() 函数把 class 文件中的常量池转换成运行时常量池。这个函数稍微有点复杂，主体代码如下：

```
func newConstantPool(class *Class, cfCp classfile.ConstantPool) *ConstantPool {
    cpCount := len(cfCp)
    consts := make([]Constant, cpCount)
    rtCp := &ConstantPool{class, consts}

    for i := 1; i < cpCount; i++ {
        cpInfo := cfCp[i]
        switch cpInfo.(type) {
        ...
        }
    }

    return rtCp
}
```

其实也不难理解，核心逻辑就是把 []classfile.ConstantInfo 转换成 []heap.Constant。具体常量的转换在 switch-case 中，我们分几次来看。

最简单的是 int 或 float 型常量，直接取出常量值，放进 consts 中即可。

```
switch cpInfo.(type) {
case *classfile.ConstantIntegerInfo:
    intInfo := cpInfo.(*classfile.ConstantIntegerInfo)
    consts[i] = intInfo.Value() // int32
case *classfile.ConstantFloatInfo:
    floatInfo := cpInfo.(*classfile.ConstantFloatInfo)
    consts[i] = floatInfo.Value() // float32
```

如果是 long 或 double 型常量，也是直接提取常量值放进 consts 中。但是要注意，这两种类型的常量在常量池中都是占据两个位置，所以索引要特殊处理，代码如下：

```
case *classfile.ConstantLongInfo:
    longInfo := cpInfo.(*classfile.ConstantLongInfo)
    consts[i] = longInfo.Value() // int64
    i++
case *classfile.ConstantDoubleInfo:
```

```
        doubleInfo := cpInfo.(*classfile.ConstantDoubleInfo)
        consts[i] = doubleInfo.Value() // float64
        i++
```

如果是字符串常量，直接取出 Go 语言字符串，放进 consts 中，代码如下：

```
    case *classfile.ConstantStringInfo:
        stringInfo := cpInfo.(*classfile.ConstantStringInfo)
        consts[i] = stringInfo.String() // string
```

还剩下 4 种类型的常量需要处理，分别是类、字段、方法和接口方法的符号引用。后面的章节会详细介绍这 4 种符号引用，下面是剩下的代码。

```
    case *classfile.ConstantClassInfo:
        classInfo := cpInfo.(*classfile.ConstantClassInfo)
        consts[i] = newClassRef(rtCp, classInfo) // 见 6.2.1 小节
    case *classfile.ConstantFieldrefInfo:
        fieldrefInfo := cpInfo.(*classfile.ConstantFieldrefInfo)
        consts[i] = newFieldRef(rtCp, fieldrefInfo) // 见 6.2.2 小节
    case *classfile.ConstantMethodrefInfo:
        methodrefInfo := cpInfo.(*classfile.ConstantMethodrefInfo)
        consts[i] = newMethodRef(rtCp, methodrefInfo) // 见 6.2.3 小节
    case *classfile.ConstantInterfaceMethodrefInfo:
        methodrefInfo := cpInfo.(*classfile.ConstantInterfaceMethodrefInfo)
        consts[i] = newInterfaceMethodRef(rtCp, methodrefInfo) // 见 6.2.4 小节
```

基本类型常量的使用请参考 6.4 节。

6.2.1 类符号引用

因为 4 种类型的符号引用有一些共性，所以仍然使用继承来减少重复代码。在 ch06\rtda\heap 目录下创建 cp_symref.go 文件，在其中定义 SymRef 结构体，代码如下：

```
package heap

// symbolic reference
type SymRef struct {
    cp          *ConstantPool
    className   string
    class       *Class
}
```

cp 字段存放符号引用所在的运行时常量池指针，这样就可以通过符号引用访问到运行时常量池，进一步又可以访问到类数据。className 字段存放类的完全限定名。class 字段缓存解析后的类结构体指针，这样类符号引用只需要解析一次就可以了，后续可以直接使

用缓存值。对于类符号引用，只要有类名，就可以解析符号引用。对于字段，首先要解析类符号引用得到类数据，然后用字段名和描述符查找字段数据。方法符号引用的解析过程和字段符号引用类似。

SymRef 定义好了，接下来在 ch06\rtda\heap 目录下创建 cp_classref.go 文件，在其中定义 ClassRef 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type ClassRef struct {
    SymRef
}

func newClassRef(cp *ConstantPool,
      classInfo *classfile.ConstantClassInfo) *ClassRef {...}
```

ClassRef 继承了 SymRef，但是并没有添加任何字段。newClassRef() 函数根据 class 文件中存储的类常量创建 ClassRef 实例，代码如下：

```
func newClassRef(cp *ConstantPool,
      classInfo *classfile.ConstantClassInfo) *ClassRef {
    ref := &ClassRef{}
    ref.cp = cp
    ref.className = classInfo.Name()
    return ref
}
```

类符号引用的解析将在 6.5.2 节讨论。

6.2.2 字段符号引用

在 6.1.2 节中，定义了 ClassMember 结构体来存放字段和方法共有的信息。类似地，本节定义 MemberRef 结构体来存放字段和方法符号引用共有的信息。在 ch06\rtda\heap 目录下创建 cp_memberref.go 文件，在其中定义 MemberRef 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type MemberRef struct {
    SymRef
    name        string
```

```
    descriptor     string
}
func (self *MemberRef) copyMemberRefInfo(
      refInfo *classfile.ConstantMemberrefInfo) {...}
```

读者可能会有疑问：在Java中，我们并不能在同一个类中定义名字相同，但类型不同的两个字段，那么字段符号引用为什么还要存放字段描述符呢？答案是，这只是Java语言的限制，而不是Java虚拟机规范的限制。也就是说，站在Java虚拟机的角度，一个类是完全可以有多个同名字段的，只要它们的类型互不相同就可以。copyMemberRefInfo()方法从class文件内存储的字段或方法常量中提取数据，代码如下：

```
func (self *MemberRef) copyMemberRefInfo(refInfo *classfile.ConstantMemberrefInfo) {
    self.className = refInfo.ClassName()
    self.name, self.descriptor = refInfo.NameAndDescriptor()
}
```

MemberRef定义好了，接下来在ch06\rtda\heap目录下创建cp_fieldref.go文件，在其中定义FieldRef结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type FieldRef struct {
    MemberRef
    field *Field
}

func newFieldRef(cp *ConstantPool,
      refInfo *classfile.ConstantFieldrefInfo) *FieldRef {...}
```

field字段缓存解析后的字段指针，newFieldRef()方法创建FieldRef实例，代码如下：

```
func newFieldRef(cp *ConstantPool,
      refInfo *classfile.ConstantFieldrefInfo) *FieldRef {
    ref := &FieldRef{}
    ref.cp = cp
    ref.copyMemberRefInfo(&refInfo.ConstantMemberrefInfo)
    return ref
}
```

字段符号引用的解析将在6.5.2节讨论。

6.2.3 方法符号引用

在 ch06\rtda\heap 目录下创建 cp_methodref.go 文件，在其中定义 MethodRef 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type MethodRef struct {
    MemberRef
    method *Method
}

func newMethodRef(cp *ConstantPool,
        refInfo *classfile.ConstantMethodrefInfo) *MethodRef {
    ref := &MethodRef{}
    ref.cp = cp
    ref.copyMemberRefInfo(&refInfo.ConstantMemberrefInfo)
    return ref
}
```

上面的代码和字段符号引用大同小异，这里就不多解释了。方法符号引用的解析将在第 7 章讨论方法调用时详细介绍。

6.2.4 接口方法符号引用

在 ch06\rtda\heap 目录下创建 cp_interface_methodref.go 文件，在其中定义 InterfaceMethodRef 结构体，代码如下：

```
package heap

import "jvmgo/ch06/classfile"

type InterfaceMethodRef struct {
    MemberRef
    method *Method
}

func newInterfaceMethodRef(cp *ConstantPool,
        refInfo *classfile.ConstantInterfaceMethodrefInfo) *InterfaceMethodRef {
    ref := &InterfaceMethodRef{}
    ref.cp = cp
    ref.copyMemberRefInfo(&refInfo.ConstantMemberrefInfo)
    return ref
}
```

代码和前面差不多，也不多解释了。接口方法符号引用的解析同样会在第7章详细介绍。到此为止，所有的符号引用都已经定义好了，它们的继承结构如图6-2所示。

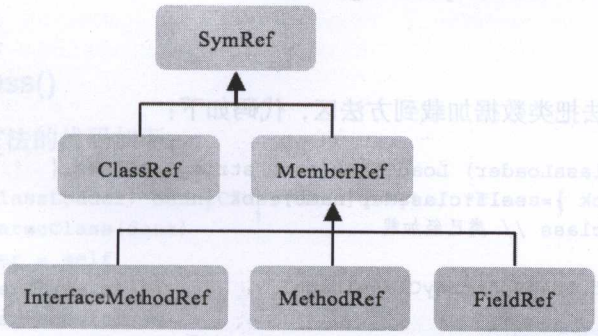


图6-2 符号引用结构体继承关系图

6.3 类加载器

Java虚拟机的类加载系统十分复杂，本节将初步实现一个简化版的类加载器，后面的章节中还会对它进行扩展。在ch06/rtda/heap目录下创建class_loader.go文件，在其中定义ClassLoader结构体，代码如下：

```
package heap

import "fmt"
import "jvmgo/ch06/classfile"
import "jvmgo/ch06/classpath"

type ClassLoader struct {
    cp       *classpath.Classpath
    classMap map[string]*Class // loaded classes
}

func NewClassLoader(cp *classpath.Classpath) *ClassLoader {...}
func (self *ClassLoader) LoadClass(name string) *Class {...}
```

ClassLoader依赖Classpath来搜索和读取class文件，cp字段保存Classpath指针。classMap字段记录已经加载的类数据，key是类的完全限定名。在前面讨论中，方法区一直只是个抽象的概念，现在可以把classMap字段当作方法区的具体实现。NewClassLoader()函数创建ClassLoader实例，代码比较简单，如下所示：

```
func NewClassLoader(cp *classpath.Classpath) *ClassLoader {
    return &ClassLoader{
        cp:       cp,
        classMap:   make(map[string]*Class),
    }
}
```

LoadClass() 方法把类数据加载到方法区，代码如下：

```
func (self *ClassLoader) LoadClass(name string) *Class {
    if class, ok := self.classMap[name]; ok {
        return class // 类已经加载
    }
    return self.loadNonArrayClass(name)
}
```

先查找 classMap，看类是否已经被加载。如果是，直接返回类数据，否则调用 loadNonArrayClass() 方法加载类。数组类和普通类有很大的不同，它的数据并不是来自 class 文件，而是由 Java 虚拟机在运行期间生成。本章暂不考虑数组类的加载，留到第 8 章详细讨论。loadNonArrayClass() 方法的代码如下：

```
func (self *ClassLoader) loadNonArrayClass(name string) *Class {
    data, entry := self.readClass(name)
    class := self.defineClass(data)
    link(class)
    fmt.Printf("[Loaded %s from %s]\n", name, entry)
    return class
}
```

可以看到，类的加载大致可以分为三个步骤：首先找到 class 文件并把数据读取到内存；然后解析 class 文件，生成虚拟机可以使用的类数据，并放入方法区；最后进行链接。下面分别讨论这三个步骤。

6.3.1 readClass()

readClass() 方法的代码如下：

```
func (self *ClassLoader) readClass(name string) ([]byte, classpath.Entry) {
    data, entry, err := self.cp.ReadClass(name)
    if err != nil {
        panic("java.lang.ClassNotFoundException: " + name)
    }
    return data, entry
}
```

readClass() 方法只是调用了 Classpath 的 ReadClass() 方法，并进行了错误处理。需要解释一下它的返回值。为了打印类加载信息，把最终加载 class 文件的类路径项也返回给了调用者。

6.3.2 defineClass()

defineClass() 方法的代码如下：

```
func (self *ClassLoader) defineClass(data []byte) *Class {
    class := parseClass(data)
    class.loader = self
    resolveSuperClass(class)
    resolveInterfaces(class)
    self.classMap[class.name] = class
    return class
}
```

defineClass() 方法首先调用 parseClass() 函数把 class 文件数据转换成 Class 结构体。Class 结构体的 superClass 和 interfaces 字段存放超类名和直接接口表，这些类名其实都是符号引用。根据 Java 虚拟机规范的 5.3.5 节，调用 resolveSuperClass() 和 resolveInterfaces() 函数解析这些类符号引用。下面是 parseClass() 函数的代码。

```
func parseClass(data []byte) *Class {
    cf, err := classfile.Parse(data)
    if err != nil {
        panic("java.lang.ClassFormatError")
    }
    return newClass(cf) // 见 6.1.1 小节
}
```

resolveSuperClass() 函数的代码如下：

```
func resolveSuperClass(class *Class) {
    if class.name != "java/lang/Object" {
        class.superClass = class.loader.LoadClass(class.superClassName)
    }
}
```

再复习一下：除 java.lang.Object 以外，所有的类都有且仅有一个超类。因此，除非是 Object 类，否则需要递归调用 LoadClass() 方法加载它的超类。与此类似，resolveInterfaces() 函数递归调用 LoadClass() 方法加载类的每一个直接接口，代码如下：

```
func resolveInterfaces(class *Class) {
    interfaceCount := len(class.interfaceNames)
    if interfaceCount > 0 {
        class.interfaces = make([]*Class, interfaceCount)
        for i, interfaceName := range class.interfaceNames {
            class.interfaces[i] = class.loader.LoadClass(interfaceName)
        }
    }
}
```

6.3.3 link()

类的链接分为验证和准备两个必要阶段，link() 方法的代码如下：

```
func link(class *Class) {
    verify(class)
    prepare(class)
}
```

为了确保安全性，Java 虚拟机规范要求在执行类的任何代码之前，对类进行严格的验证。由于篇幅的原因，本书忽略验证过程。Java 虚拟机规范 4.10 节详细介绍了类的验证算法，感兴趣的读者可以尝试自己实现。verify() 函数空空如也，代码如下：

```
func verify(class *Class) {
    // todo
}
```

准备阶段主要是给类变量分配空间并给予初始值，prepare() 函数推迟到 6.4 节再介绍。

6.4 对象、实例变量和类变量

在第 4 章中，定义了 LocalVars 结构体，用来表示局部变量表。从逻辑上来看，LocalVars 实例就像一个数组，这个数组的每一个元素都足够容纳一个 int、float 或引用值。要放入 double 或者 long 值，需要相邻的两个元素。这个结构体不是正好也可以用来表示类变量和实例变量吗？

没错！但是，由于 rtda 包已经依赖了 heap 包，而 Go 语言的包又不能相互依赖，所以 heap 包中的 go 文件是无法导入 rtda 包的，否则 Go 编译器就会报错。为了解决这个问题，只好容忍一些重复代码的存在。在 ch06\rtda\heap 目录下创建 slots.go 文件，把 slot.go 和 local_vars.go 文件中的内容拷贝进来，然后在此基础上修改，代码如下：

```
package heap

import "math"

type Slot struct {
    num int32
    ref *Object
}

type Slots []Slot
```

函数和方法的内容都没什么变化，为了节约篇幅，就不给出详细代码了，下面是列表。

```
func newSlots(slotCount uint) Slots {...}
func (self Slots) SetInt(index uint, val int32) {...}
func (self Slots) GetInt(index uint) int32 {...}
func (self Slots) SetFloat(index uint, val float32) {...}
func (self Slots) GetFloat(index uint) float32 {...}
func (self Slots) SetLong(index uint, val int64) {...}
func (self Slots) GetLong(index uint) int64 {...}
func (self Slots) SetDouble(index uint, val float64) {...}
func (self Slots) GetDouble(index uint) float64 {...}
func (self Slots) SetRef(index uint, ref *Object) {...}
func (self Slots) GetRef(index uint) *Object {...}
```

Slots 结构体准备就绪，可以使用了。Class 结构体早在 6.1 节就定义好了，代码如下：

```
type Class struct {
    ...
    staticVars *Slots
}
```

打开 ch06\rtda\heap\object.go 文件，给 Object 结构体添加两个字段，一个存放对象的 Class 指针，一个存放实例变量，代码如下：

```
type Object struct {
    class  *Class
    fields Slots
}
```

接下来的问题是，如何知道静态变量和实例变量需要多少空间，以及哪个字段对应 Slots 中的哪个位置呢？

第一个问题比较好解决，只要数一下类的字段即可。假设某个类有 m 个静态字段和 n 个实例字段，那么静态变量和实例变量所需的空间大小就分别是 m' 和 n'。这里要注意两点。首先，类是可以继承的。也就是说，在数实例变量时，要递归地数超类的实例变量；

其次，long 和 double 字段都占据两个位置，所以 m'>=m，n'>=n。

第二个问题也不算难，在数字段时，给字段按顺序编上号就可以了。这里有三点需要要注意。首先，静态字段和实例字段要分开编号，否则会混乱。其次，对于实例字段，一定要从继承关系的最顶端，也就是 java.lang.Object 开始编号，否则也会混乱。最后，编号时也要考虑 long 和 double 类型。

打开 field.go 文件，给 Field 结构体加上 slotId 字段，代码如下：

```
type Field struct {
    ClassMember
    slotId uint
}
```

打开 class_loader.go 文件，在其中定义 prepare() 函数，代码如下：

```
func prepare(class *Class) {
    calcInstanceFieldSlotIds(class)
    calcStaticFieldSlotIds(class)
    allocAndInitStaticVars(class)
}
```

calcInstanceFieldSlotIds() 函数计算实例字段的个数，同时给它们编号，代码如下：

```
func calcInstanceFieldSlotIds(class *Class) {
    slotId := uint(0)
    if class.superClass != nil {
        slotId = class.superClass.instanceSlotCount
    }
    for _, field := range class.fields {
        if !field.IsStatic() {
            field.slotId = slotId
            slotId++
            if field.isLongOrDouble() {
                slotId++
            }
        }
    }
    class.instanceSlotCount = slotId
}
```

calcStaticFieldSlotIds() 函数计算静态字段的个数，同时给它们编号，代码如下：

```
func calcStaticFieldSlotIds(class *Class) {
    slotId := uint(0)
    for _, field := range class.fields {
```

```
        if field.IsStatic() {
            field.slotId = slotId
            slotId++
            if field.isLongOrDouble() {
                slotId++
            }
        }
    }
    class.staticSlotCount = slotId
}
```

Field 结构体的 isLongOrDouble() 方法返回字段是否是 long 或 double 类型，代码如下：

```
func (self *Field) isLongOrDouble() bool {
    return self.descriptor == "J" || self.descriptor == "D"
}
```

allocAndInitStaticVars() 函数给类变量分配空间，然后给它们赋予初始值，代码如下：

```
func allocAndInitStaticVars(class *Class) {
    class.staticVars = newSlots(class.staticSlotCount)
    for _, field := range class.fields {
        if field.IsStatic() && field.IsFinal() {
            initStaticFinalVar(class, field)
        }
    }
}
```

因为 Go 语言会保证新创建的 Slot 结构体有默认值（num 字段是 0，ref 字段是 nil），而浮点数 0 编码之后和整数 0 相同，所以不用做任何操作就可以保证静态变量有默认初始值（数字类型是 0，引用类型是 null）。如果静态变量属于基本类型或 String 类型，有 final 修饰符，且它的值在编译期已知，则该值存储在 class 文件常量池中。initStaticFinalVar() 函数从常量池中加载常量值，然后给静态变量赋值，代码如下：

```
func initStaticFinalVar(class *Class, field *Field) {
    vars := class.staticVars
    cp := class.constantPool
    cpIndex := field.ConstValueIndex()
    slotId := field.SlotId()

    if cpIndex > 0 {
        switch field.Descriptor() {
        case "Z", "B", "C", "S", "I":
            val := cp.GetConstant(cpIndex).(int32)
            vars.SetInt(slotId, val)
        case "J":
```

```
            val := cp.GetConstant(cpIndex).(int64)
            vars.SetLong(slotId, val)
        case "F":
            val := cp.GetConstant(cpIndex).(float32)
            vars.SetFloat(slotId, val)
        case "D":
            val := cp.GetConstant(cpIndex).(float64)
            vars.SetDouble(slotId, val)
        case "Ljava/lang/String;":
            panic("todo") // 在第 8 章实现
        }
    }
}
```

字符串常量将在第 8 章讨论，这里先调用 panic() 函数终止程序执行。需要给 Field 结构体添加 constValueIndex 字段，代码如下：

```
type Field struct {
    ClassMember
    constValueIndex     uint
    slotId              uint
}
```

修改 newFields() 方法，从字段属性表中读取 constValueIndex，代码改动如下：

```
func newFields(class *Class, cfFields []*classfile.MemberInfo) []*Field {
    fields := make([]*Field, len(cfFields))
    for i, cfField := range cfFields {
        fields[i] = &Field{}
        fields[i].class = class
        fields[i].copyMemberInfo(cfField)
        fields[i].copyAttributes(cfField)
    }
    return fields
}
```

copyAttributes() 方法的代码如下：

```
func (self *Field) copyAttributes(cfField *classfile.MemberInfo) {
    if valAttr := cfField.ConstantValueAttribute(); valAttr != nil {
        self.constValueIndex = uint(valAttr.ConstantValueIndex())
    }
}
```

MemberInfo 结构体的 ConstantValueIndex() 方法在 ch06\classfile\member_info.go 文件中，代码如下：

```
func (self *MemberInfo) ConstantValueAttribute() *ConstantValueAttribute {
    for _, attrInfo := range self.attributes {
        switch attrInfo.(type) {
        case *ConstantValueAttribute:
            return attrInfo.(*ConstantValueAttribute)
        }
    }
    return nil
}
```

6.5 类和字段符号引用解析

本节讨论类符号引用和字段符号引用的解析，方法符号引用的解析将在第7章讨论。

6.5.1 类符号引用解析

打开cp_symref.go文件，在其中定义ResolvedClass()方法，代码如下：

```
func (self *SymRef) ResolvedClass() *Class {
    if self.class == nil {
        self.resolveClassRef()
    }
    return self.class
}
```

如果类符号引用已经解析，ResolvedClass()方法直接返回类指针，否则调用resolveClassRef()方法进行解析。Java虚拟机规范5.4.3.1节给出了类符号引用的解析步骤，resolveClassRef()方法就是按照这个步骤编写的（有一些简化），代码如下：

```
func (self *SymRef) resolveClassRef() {
    d := self.cp.class
    c := d.loader.LoadClass(self.className)
    if !c.isAccessibleTo(d) {
        panic("java.lang.IllegalAccessError")
    }
    self.class = c
}
```

通俗地讲，如果类D通过符号引用N引用类C的话，要解析N，先用D的类加载器加载C，然后检查D是否有权限访问C，如果没有，则抛出IllegalAccessError异常。Java虚拟机规范5.4.4节给出了类的访问控制规则，把这个规则翻译成Class结构体的isAccessibleTo()方法，代码如下（在class.go文件中）：

```
func (self *Class) isAccessibleTo(other *Class) bool {
    return self.IsPublic() || self.getPackageName() == other.getPackageName()
}
```

也就是说，如果类 D 想访问类 C，需要满足两个条件之一：C 是 public，或者 C 和 D 在同一个运行时包内。第 11 章再讨论运行时包，这里先简单按照包名来检查。getPackageName() 方法的代码如下（也在 class.go 文件中）：

```
func (self *Class) getPackageName() string {
    if i := strings.LastIndex(self.name, "/"); i >= 0 {
        return self.name[:i]
    }
    return ""
}
```

比如类名是 java/lang/Object，则它的包名就是 java/lang。如果类定义在默认包中，它的包名是空字符串。

6.5.2 字段符号引用解析

打开 cp_fieldref.go 文件，在其中定义 ResolvedField() 方法，代码如下：

```
func (self *FieldRef) ResolvedField() *Field {
    if self.field == nil {
        self.resolveFieldRef()
    }
    return self.field
}
```

ResolvedField() 方法与 ResolvedClass() 方法大同小异，就不多解释了。Java 虚拟机规范 5.4.3.2 节给出了字段符号引用的解析步骤，把它翻译成 resolveFieldRef() 方法，代码如下：

```
func (self *FieldRef) resolveFieldRef() {
    d := self.cp.class
    c := self.ResolvedClass()
    field := lookupField(c, self.name, self.descriptor)

    if field == nil {
        panic("java.lang.NoSuchFieldError")
    }
    if !field.isAccessibleTo(d) {
        panic("java.lang.IllegalAccessError")
    }

    self.field = field
```

```
}
```

如果类 D 想通过字段符号引用访问类 C 的某个字段，首先要解析符号引用得到类 C，然后根据字段名和描述符查找字段。如果字段查找失败，则虚拟机抛出 NoSuchFieldError 异常。如果查找成功，但 D 没有足够的权限访问该字段，则虚拟机抛出 IllegalAccessError 异常。字段查找步骤在 lookupField() 函数中，代码如下：

```
func lookupField(c *Class, name, descriptor string) *Field {
    for _, field := range c.fields {
        if field.name == name && field.descriptor == descriptor {
            return field
        }
    }
    for _, iface := range c.interfaces {
        if field := lookupField(iface, name, descriptor); field != nil {
            return field
        }
    }
    if c.superClass != nil {
        return lookupField(c.superClass, name, descriptor)
    }
    return nil
}
```

首先在 C 的字段中查找。如果找不到，在 C 的直接接口递归应用这个查找过程。如果还找不到的话，在 C 的超类中递归应用这个查找过程。如果仍然找不到，则查找失败。Java 虚拟机规范 5.4.4 节也给出了字段的访问控制规则。这个规则同样也适用于方法，所以把它（略做简化）实现成 ClassMember 结构体的 isAccessibleTo() 方法，代码如下（在 class_member.go 文件中）：

```
func (self *ClassMember) isAccessibleTo(d *Class) bool {
    if self.IsPublic() {
        return true
    }
    c := self.class
    if self.IsProtected() {
        return d == c || d.isSubClassOf(c) ||
            c.getPackageName() == d.getPackageName()
    }
    if !self.IsPrivate() {
        return c.getPackageName() == d.getPackageName()
    }
    return d == c
}
```

用通俗的语言描述字段访问规则。如果字段是 public，则任何类都可以访问。如果字段是 protected，则只有子类和同一个包下的类可以访问。如果字段有默认访问权限（非 public，非 protected，也非 privated），则只有同一个包下的类可以访问。否则，字段是 private，只有声明这个字段的类才能访问。

6.6 类和对象相关指令

本节将实现 10 条类和对象相关的指令。new 指令用来创建类实例；putstatic 和 getstatic 指令用于存取静态变量；putfield 和 getfield 用于存取实例变量；instanceof 和 checkcast 指令用于判断对象是否属于某种类型；ldc 系列指令把运行时常量池中的常量推到操作数栈顶。下面的 Java 代码演示了这些指令的用处。

```
public class MyObject {
    public static int staticVar;
    public int instanceVar;

    public static void main(String[] args) {
        int x = 32768;                          // ldc
        MyObject myObj = new MyObject();        // new
        MyObject.staticVar = x;                 // putstatic
        x = MyObject.staticVar;                 // getstatic
        myObj.instanceVar = x;                  // putfield
        x = myObj.instanceVar;                  // getfield
        Object obj = myObj;
        if (obj instanceof MyObject) {          // instanceof
            myObj = (MyObject) obj;             // checkcast
        }
    }
}
```

上面提到的指令除 ldc 以外，都属于引用类指令，在 ch06\instructions 目录下创建 references 子目录来存放引用类指令。首先实现 new 指令。

6.6.1 new 指令

注意，new 指令专门用来创建类实例。数组由专门的指令创建，在第 8 章中实现数组和数组相关指令。在 ch06\instructions\references 目录下创建 new.go 文件，在其中实现 new 指令，代码如下：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"
// Create new object
type NEW struct{ base.Index16Instruction }
```

new 指令的操作数是一个 uint16 索引，来自字节码。通过这个索引，可以从当前类的运行时常量池中找到一个类符号引用。解析这个类符号引用，拿到类数据，然后创建对象，并把对象引用推入栈顶，new 指令的工作就完成了。Execute() 方法的代码如下：

```
func (self *NEW) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    classRef := cp.GetConstant(self.Index).(*heap.ClassRef)
    class := classRef.ResolvedClass()

    if class.IsInterface() || class.IsAbstract() {
        panic("java.lang.InstantiationError")
    }

    ref := class.NewObject()
    frame.OperandStack().PushRef(ref)
}
```

因为接口和抽象类都不能实例化，所以如果解析后的类是接口或抽象类，按照 Java 虚拟机规范规定，需要抛出 InstantiationError 异常。另外，如果解析后的类还没有初始化，则需要先初始化类。在第 7 章实现方法调用之后会详细讨论类的初始化，这里暂时先忽略。Class 结构体的 NewObject() 方法如下（在 class.go 文件中）：

```
func (self *Class) NewObject() *Object {
    return newObject(self)
}
```

这里只是调用了 Object 结构体的 newObject() 方法，代码如下（在 object.go 中）：

```
func newObject(class *Class) *Object {
    return &Object{
        class:  class,
        fields: newSlots(class.instanceSlotCount),
    }
}
```

新创建对象的实例变量都应该赋好初始值，不过并不需要做额外的工作，具体原因前

面已经讨论过，此处不再赘述。new 指令实现好了，下面看看如何存取类的静态变量。

6.6.2 putstatic 和 getstatic 指令

在 references 目录下创建 putstatic.go 文件，在其中实现 putstatic 指令，代码如下：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"

// Set static field in class
type PUT_STATIC struct{ base.Index16Instruction }
```

putstatic 指令给类的某个静态变量赋值，它需要两个操作数。第一个操作数是 uint16 索引，来自字节码。通过这个索引可以从当前类的运行时常量池中找到一个字段符号引用，解析这个符号引用就可以知道要给类的哪个静态变量赋值。第二个操作数是要赋给静态变量的值，从操作数栈中弹出。Execute() 方法稍微有些复杂，分三部分介绍：

```
func (self *PUT_STATIC) Execute(frame *rtda.Frame) {
    currentMethod := frame.Method()
    currentClass := currentMethod.Class()
    cp := currentClass.ConstantPool()
    fieldRef := cp.GetConstant(self.Index).(*heap.FieldRef)
    field := fieldRef.ResolvedField()
    class := field.Class()
```

先拿到当前方法、当前类和当前常量池，然后解析字段符号引用。如果声明字段的类还没有被初始化，则需要先初始化该类，这部分逻辑将在第 7 章实现。继续看代码：

```
    if !field.IsStatic() {
        panic("java.lang.IncompatibleClassChangeError")
    }
    if field.IsFinal() {
        if currentClass != class || currentMethod.Name() != "<clinit>" {
            panic("java.lang.IllegalAccessError")
        }
    }
```

如果解析后的字段是实例字段而非静态字段，则抛出 IncompatibleClassChangeError 异常。如果是 final 字段，则实际操作的是静态常量，只能在类初始化方法中给它赋值。否则，会抛出 IllegalAccessError 异常。类初始化方法由编译器生成，名字是 <clinit>，具体

请看第7章。继续看代码：

```
    descriptor := field.Descriptor()
    slotId := field.SlotId()
    slots := class.StaticVars()
    stack := frame.OperandStack()

    switch descriptor[0] {
    case 'Z', 'B', 'C', 'S', 'I': slots.SetInt(slotId, stack.PopInt())
    case 'F': slots.SetFloat(slotId, stack.PopFloat())
    case 'J': slots.SetLong(slotId, stack.PopLong())
    case 'D': slots.SetDouble(slotId, stack.PopDouble())
    case 'L', '[': slots.SetRef(slotId, stack.PopRef())
    }
}
```

根据字段类型从操作数栈中弹出相应的值，然后赋给静态变量。至此，putstatic 指令就解释完毕了。getstatic 指令和 putstatic 正好相反，它取出类的某个静态变量值，然后推入栈顶。在 references 目录下创建 getstatic.go 文件，在其中实现 getstatic 指令，代码如下：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"

// Get static field from class
type GET_STATIC struct{ base.Index16Instruction }
```

getstatic 指令只需要一个操作数：uint16 常量池索引，用法和 putstatic 一样，代码如下：

```
func (self *GET_STATIC) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    fieldRef := cp.GetConstant(self.Index).(*heap.FieldRef)
    field := fieldRef.ResolvedField()
    class := field.Class()

    if !field.IsStatic() {
        panic("java.lang.IncompatibleClassChangeError")
    }
```

如果解析后的字段不是静态字段，也要抛出 IncompatibleClassChangeError 异常。如果声明字段的类还没有初始化好，也需要先初始化。getstatic 只是读取静态变量的值，自然也就不用管它是否是 final 了。继续看剩下的代码：

```
    descriptor := field.Descriptor()
```

```
    slotId := field.SlotId()
    slots := class.StaticVars()
    stack := frame.OperandStack()
    switch descriptor[0] {
    case 'Z', 'B', 'C', 'S', 'I': stack.PushInt(slots.GetInt(slotId))
    case 'F': stack.PushFloat(slots.GetFloat(slotId))
    case 'J': stack.PushLong(slots.GetLong(slotId))
    case 'D': stack.PushDouble(slots.GetDouble(slotId))
    case 'L', '[': stack.PushRef(slots.GetRef(slotId))
    }
}
```

根据字段类型，从静态变量中取出相应的值，然后推入操作数栈顶。至此，getstatic 指令也解释完毕了。下面介绍如何存取对象的实例变量。

6.6.3 putfield 和 getfield 指令

在 references 目录下创建 putfield.go 文件，在其中实现 putfield 指令，代码如下：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"

// Set field in object
type PUT_FIELD struct{ base.Index16Instruction }
```

putfield 指令给实例变量赋值，它需要三个操作数。前两个操作数是常量池索引和变量值，用法和 putstatic 一样。第三个操作数是对象引用，从操作数栈中弹出。同样分三次来介绍 putfield 指令的 Execute() 方法，第一部分代码如下：

```
func (self *PUT_FIELD) Execute(frame *rtda.Frame) {
    currentMethod := frame.Method()
    currentClass := currentMethod.Class()
    cp := currentClass.ConstantPool()
    fieldRef := cp.GetConstant(self.Index).(*heap.FieldRef)
    field := fieldRef.ResolvedField()
```

基本上和 putstatic 一样，这里就不多解释了。看下一部分：

```
if field.IsStatic() {
    panic("java.lang.IncompatibleClassChangeError")
}
```

```
if field.IsFinal() {
    if currentClass != field.Class() || currentMethod.Name() != "<init>" {
        panic("java.lang.IllegalAccessError")
    }
}
```

看起来也和 putstatic 差不多，但有两点不同（在代码中已经加粗）。第一，解析后的字段必须是实例字段，否则抛出 IncompatibleClassChangeError。第二，如果是 final 字段，则只能在构造函数中初始化，否则抛出 IllegalAccessError。在第 7 章会介绍构造函数。下面看剩下的代码：

```
    descriptor := field.Descriptor()
    slotId := field.SlotId()
    stack := frame.OperandStack()

    switch descriptor[0] {
    case 'Z', 'B', 'C', 'S', 'I':
        val := stack.PopInt()
        ref := stack.PopRef()
        if ref == nil {
            panic("java.lang.NullPointerException")
        }
        ref.Fields().SetInt(slotId, val)
    case 'F': ...
    case 'J': ...
    case 'D': ...
    case 'L', '[': ...
    }
}
```

先根据字段类型从操作数栈中弹出相应的变量值，然后弹出对象引用。如果引用是 null，需要抛出著名的空指针异常（NullPointerException），否则通过引用给实例变量赋值。其他的 case 语句和第一个大同小异，为了节约篇幅，省略了详细代码。

putfield 指令解释完毕，下面来看 getfield 指令。在 references 目录下创建 getfield.go 文件，在其中实现 getfield 指令，代码如下：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"

// Fetch field from object
type GET_FIELD struct{ base.Index16Instruction }
```

getfield 指令获取对象的实例变量值，然后推入操作数栈，它需要两个操作数。第一个操作数是 uint16 索引，用法和前面三个指令一样。第二个操作数是对象引用，用法和 putfield 一样。下面看看 getfield 指令的 Execute 方法 ()，第一部分代码如下：

```
func (self *GET_FIELD) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    fieldRef := cp.GetConstant(self.Index).(*heap.FieldRef)
    field := fieldRef.ResolvedField()
    if field.IsStatic() {
        panic("java.lang.IncompatibleClassChangeError")
    }
```

先是字段符号引用解析。这部分逻辑我们已经很熟悉了，不多解释。下面是第二部分代码：

```
stack := frame.OperandStack()
ref := stack.PopRef()
if ref == nil {
    panic("java.lang.NullPointerException")
}
```

弹出对象引用，如果是 null，则抛出 NullPointerException。剩下的代码如下：

```
    descriptor := field.Descriptor()
    slotId := field.SlotId()
    slots := ref.Fields()

    switch descriptor[0] {
    case 'Z', 'B', 'C', 'S', 'I': stack.PushInt(slots.GetInt(slotId))
    case 'F': stack.PushFloat(slots.GetFloat(slotId))
    case 'J': stack.PushLong(slots.GetLong(slotId))
    case 'D': stack.PushDouble(slots.GetDouble(slotId))
    case 'L', '[': stack.PushRef(slots.GetRef(slotId))
    }
}
```

根据字段类型，获取相应的实例变量值，然后推入操作数栈。至此，getfield 指令也解释完毕了。下面讨论 instanceof 和 checkcast 指令。

6.6.4 instanceof 和 checkcast 指令

instanceof 指令判断对象是否是某个类的实例（或者对象的类是否实现了某个接口），并把结果推入操作数栈。在 references 目录下创建 instanceof.go 文件，在其中实现 instanceof 指令，代码如下：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"

// Determine if object is of given type
type INSTANCE_OF struct{ base.Index16Instruction }
```

instanceof指令需要两个操作数。第一个操作数是uint16索引，从方法的字节码中获取，通过这个索引可以从当前类的运行时常量池中找到一个类符号引用。第二个操作数是对象引用，从操作数栈中弹出。instanceof指令的Execute()方法如下：

```
func (self *INSTANCE_OF) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    ref := stack.PopRef()
    if ref == nil {
        stack.PushInt(0)
        return
    }

    cp := frame.Method().Class().ConstantPool()
    classRef := cp.GetConstant(self.Index).(*heap.ClassRef)
    class := classRef.ResolvedClass()
    if ref.IsInstanceOf(class) {
        stack.PushInt(1)
    } else {
        stack.PushInt(0)
    }
}
```

先弹出对象引用，如果是null，则把0推入操作数栈。用Java代码解释就是，如果引用obj是null的话，不管ClassYYY是哪种类型，下面这条if判断都是false：

```
if (obj instanceof ClassYYY) {...}
```

如果对象引用不是null，则解析类符号引用，判断对象是否是类的实例，然后把判断结果推入操作数栈。Java虚拟机规范给出了具体的判断步骤，我们在Object结构体的IsInstanceOf()方法中实现，稍后给出代码。下面来看checkcast指令。在references目录下创建checkcast.go文件，在其中实现checkcast指令，代码如下：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
```

```
import "jvmgo/ch06/rtda/heap"

// Check whether object is of given type
type CHECK_CAST struct{ base.Index16Instruction }
```

checkcast 指令和 instanceof 指令很像，区别在于：instanceof 指令会改变操作数栈（弹出对象引用，推入判断结果）；checkcast 则不改变操作数栈（如果判断失败，直接抛出 ClassCastException 异常）。checkcast 指令的 Execute() 方法如下：

```
func (self *CHECK_CAST) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    ref := stack.PopRef()
    stack.PushRef(ref)
    if ref == nil {
        return
    }

    cp := frame.Method().Class().ConstantPool()
    classRef := cp.GetConstant(self.Index).(*heap.ClassRef)
    class := classRef.ResolvedClass()
    if !ref.IsInstanceOf(class) {
        panic("java.lang.ClassCastException")
    }
}
```

先从操作数栈中弹出对象引用，再推回去，这样就不会改变操作数栈的状态。如果引用是 null，则指令执行结束。也就是说，null 引用可以转换成任何类型，否则解析类符号引用，判断对象是否是类的实例。如果是的话，指令执行结束，否则抛出 ClassCastException。instanceof 和 checkcast 指令一般都是配合使用的，像下面的 Java 代码这样：

```
if (xxx instanceof ClassYYY) {
    yyy = (ClassYYY) xxx;
    // use yyy
}
```

Object 结构体的 IsInstanceOf() 方法的代码如下（在 object.go 文件中）：

```
func (self *Object) IsInstanceOf(class *Class) bool {
    return class.isAssignableFrom(self.class)
}
```

真正的逻辑在 Class 结构体的 isAssignableFrom() 方法中，这个方法稍微有些复杂，为了避免 class.go 文件变得过长，把它写在另一个文件中。在 ch06\rtda\heap 目录下创建 class_hierarchy.go 文件，在其中定义 isAssignableFrom() 方法，代码如下：

```
func (self *Class) isAssignableFrom(other *Class) bool {
    s, t := other, self
    if s == t {
        return true
    }
    if !t.IsInterface() {
        return s.isSubClassOf(t)
    } else {
        return s.isImplements(t)
    }
}
```

也就是说，在三种情况下，S类型的引用值可以赋值给T类型：S和T是同一类型；T是类且S是T的子类；或者T是接口且S实现了T接口。这是简化版的判断逻辑，因为还没有实现数组，第8章讨论数组时会继续完善这个方法。继续编辑class_hierarchy.go文件，在其中实现isSubClassOf()方法，代码如下：

```
func (self *Class) isSubClassOf(other *Class) bool {
    for c := self.superClass; c != nil; c = c.superClass {
        if c == other {
            return true
        }
    }
    return false
}
```

判断S是否是T的子类，实际上也就是判断T是否是S的（直接或间接）超类。继续编辑class_hierarchy.go文件，在其中实现isImplements()方法，代码如下：

```
func (self *Class) isImplements(iface *Class) bool {
    for c := self; c != nil; c = c.superClass {
        for _, i := range c.interfaces {
            if i == iface || i.isSubInterfaceOf(iface) {
                return true
            }
        }
    }
    return false
}
```

判断S是否实现了T接口，就看S或S的（直接或间接）超类是否实现了某个接口T',T'要么是T，要么是T的子接口。isSubInterfaceOf()方法也在class_hierarchy.go文件中，代码如下：

```
func (self *Class) isSubInterfaceOf(iface *Class) bool {
    for _, superInterface := range self.interfaces {
        if superInterface == iface || superInterface.isSubInterfaceOf(iface) {
            return true
        }
    }
    return false
}
```

isSubInterfaceOf() 方法和 isSubClassOf() 方法类似，但是用到了递归，这里不多解释了。到此为止，instanceof 和 checkcast 指令就介绍完毕了，下面来看 ldc 指令。

6.6.5 ldc 指令

ldc 系列指令从运行时常量池中加载常量值，并把它推入操作数栈。ldc 系列指令属于常量类指令，共 3 条。其中 ldc 和 ldc_w 指令用于加载 int、float 和字符串常量，java.lang.Class 实例或者 MethodType 和 MethodHandle 实例。ldc2_w 指令用于加载 long 和 double 常量。ldc 和 ldc_w 指令的区别仅在于操作数的宽度。

本章只处理 int、float、long 和 double 常量。第 8 章实现数组和字符串之后，会进一步完善 ldc 指令，支持字符串常量的加载。第 9 章还会继续完善 ldc 指令，支持 Class 实例的加载。本书不讨论 MethodType 和 MethodHandle，感兴趣的读者请参考 Java 虚拟机规范的相关章节。

在 ch06\instructions\constants 目录下创建 ldc.go 文件，在其中定义 ldc、ldc_w 和 ldc_2w 指令，代码如下：

```
package constants

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"

type LDC struct{ base.Index8Instruction }
type LDC_W struct{ base.Index16Instruction }
type LDC2_W struct{ base.Index16Instruction }
```

ldc 和 ldc_w 指令的逻辑完全一样，在 _ldc() 函数中实现，代码如下：

```
func (self *LDC) Execute(frame *rtda.Frame) {
    _ldc(frame, self.Index)
}
func (self *LDC_W) Execute(frame *rtda.Frame) {
    _ldc(frame, self.Index)
}
```

_ldc() 函数的代码如下：

```
func _ldc(frame *rtda.Frame, index uint) {
    stack := frame.OperandStack()
    cp := frame.Method().Class().ConstantPool()
    c := cp.GetConstant(index)

    switch c.(type) {
    case int32: stack.PushInt(c.(int32))
    case float32: stack.PushFloat(c.(float32))
    // case string: 在第 8 章实现
    // case *heap.ClassRef: 在第 9 章实现
    default: panic("todo: ldc!")
    }
}
```

先从当前类的运行时常量池中取出常量。如果是 int 或 float 常量，则提取出常量值，则推入操作数栈。其他情况还无法处理，暂时调用 panic() 函数终止程序执行。ldc_2w 指令的 Execute() 方法单独实现，代码如下：

```
func (self *LDC2_W) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    cp := frame.Method().Class().ConstantPool()
    c := cp.GetConstant(self.Index)

    switch c.(type) {
    case int64: stack.PushLong(c.(int64))
    case float64: stack.PushDouble(c.(float64))
    default: panic("java.lang.ClassFormatError")
    }
}
```

代码比较简单，不多解释，这里重点说一下 Frame 结构体的 Method() 方法。为了通过 frame 变量拿到当前类的运行时常量池，给 Frame 结构体添加了 method 字段，代码如下：

```
type Frame struct {
    lower        *Frame
    localVars    LocalVars
    operandStack *OperandStack
    thread       *Thread
    method       *heap.Method
    nextPC       int
}
```

Method() 是 Getter 方法，就不给出代码了。newFrame() 函数有相应变化，代码如下：

```
func newFrame(thread *Thread, method *heap.Method) *Frame {
    return &Frame{
        thread:       thread,
        method:       method,
        localVars:    newLocalVars(method.MaxLocals()),
        operandStack: newOperandStack(method.MaxStack()),
    }
}
```

到此，类和对象相关的 10 条指令都实现好了。最后还需要修改 ch06\instructions\factory.go 文件，在其中添加这些指令的 case 语句。具体改动也比较简单，这里就不给出代码了。

6.7 测试本章代码

打开 ch06\main.go 文件，修改 import 语句，代码如下：

```
package main

import "fmt"
import "strings"
import "jvmgo/ch06/classpath"
import "jvmgo/ch06/rtda/heap"
```

main() 函数不变，删掉其他函数，然后修改 startJVM() 函数，代码如下：

```
func startJVM(cmd *Cmd) {
    cp := classpath.Parse(cmd.XjreOption, cmd.cpOption)
    classLoader := heap.NewClassLoader(cp)

    className := strings.Replace(cmd.class, ".", "/", -1)
    mainClass := classLoader.LoadClass(className)
    mainMethod := mainClass.GetMainMethod()
    if mainMethod != nil {
        interpret(mainMethod)
    } else {
        fmt.Printf("Main method not found in class %s\n", cmd.class)
    }
}
```

先创建 ClassLoader 实例，然后用它来加载主类，最后执行主类的 main() 方法。Class 结构体的 GetMainMethod() 方法如下（在 ch06\rtda\heap\class.go 文件中）：

```
func (self *Class) GetMainMethod() *Method {
    return self.getStaticMethod("main", "([Ljava/lang/String;)V")
}
```

它只是调用了 getStaticMethod() 方法而已，代码如下：

```
func (self *Class) getStaticMethod(name, descriptor string) *Method {
    for _, method := range self.methods {
        if method.IsStatic() &&
            method.name == name && method.descriptor == descriptor {

            return method
        }
    }
    return nil
}
```

接下来编辑 ch06\interpreter.go 文件，修改 import 语句，代码如下：

```
package main

import "fmt"
import "jvmgo/ch06/instructions"
import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"
```

其他函数不变，只修改 interpret() 函数，代码如下：

```
func interpret(method *heap.Method) {
    thread := rtda.NewThread()
    frame := thread.NewFrame(method)
    thread.PushFrame(frame)

    defer catchErr(frame)
    loop(thread, method.Code())
}
```

Thread 结构体的 NewFrame() 方法如下（在 ch06\rtda\thread.go 文件中）：

```
func (self *Thread) NewFrame(method *heap.Method) *Frame {
    return newFrame(self, method)
}
```

在编译本章代码之前，还需要添加两个 hack。因为对象是需要初始化的，所以每个类都至少有一个构造函数。即使用户自己不定义，编译器也会自动生成一个默认构造函数。在创建类实例时，编译器会在 new 指令的后面加入 invokespecial 指令来调用构造函数初始化对象。要到第 7 章才会实现 invokespecial 指令，为了测试 putfield 和 getfield 等指令，这里先给它一个空的实现。在 ch06\instructions\references 目录下创建 invokespecial.go 文件，

把下面的代码复制进去：

```
package references

import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"

type INVOKE_SPECIAL struct{ base.Index16Instruction }

// hack!
func (self *INVOKE_SPECIAL) Execute(frame *rtda.Frame) {
   frame.OperandStack().PopRef()
}
```

第 5 章通过打印局部变量表和操作数栈的方式观察计算结果，这样很不方便。这里用另外一个 hack 来解决这个问题。在 ch06\instructions\references 目录下创建 invokevirtual.go 文件，把下面的代码复制进去：

```
package references

import "fmt"
import "jvmgo/ch06/instructions/base"
import "jvmgo/ch06/rtda"
import "jvmgo/ch06/rtda/heap"

// Invoke instance method; dispatch based on class
type INVOKE_VIRTUAL struct{ base.Index16Instruction }

// hack!
func (self *INVOKE_VIRTUAL) Execute(frame *rtda.Frame) {
   cp := frame.Method().Class().ConstantPool()
   methodRef := cp.GetConstant(self.Index).(*heap.MethodRef)
   if methodRef.Name() == "println" {
      stack := frame.OperandStack()
      switch methodRef.Descriptor() {
      case "(Z)V": fmt.Printf("%v\n", stack.PopInt() != 0)
      case "(C)V": fmt.Printf("%c\n", stack.PopInt())
      case "(B)V": fmt.Printf("%v\n", stack.PopInt())
      case "(S)V": fmt.Printf("%v\n", stack.PopInt())
      case "(I)V": fmt.Printf("%v\n", stack.PopInt())
      case "(J)V": fmt.Printf("%v\n", stack.PopLong())
      case "(F)V": fmt.Printf("%v\n", stack.PopFloat())
      case "(D)V": fmt.Printf("%v\n", stack.PopDouble())
      default: panic("println: " + methodRef.Descriptor())
```

```
        }
        stack.PopRef()
    }
}
```

至于这两个 hack 为什么可以起作用，请阅读第 7 章，在那里会讨论方法调用和返回。有了上面的 hack，可以修改 6.6 小节开头给出的 Java 例子，添加输出语句，代码如下：

```
package jvmgo.book.ch06;

public class MyObject {
    public static int staticVar;
    public int instanceVar;

    public static void main(String[] args) {
        int x = 32768;                      // ldc
        MyObject myObj = new MyObject();    // new
        MyObject.staticVar = x;             // putstatic
        x = MyObject.staticVar;             // getstatic
        myObj.instanceVar = x;              // putfield
        x = myObj.instanceVar;              // getfield
        Object obj = myObj;
        if (obj instanceof MyObject) {      // instanceof
            myObj = (MyObject) obj;         // checkcast
            System.out.println(myObj.instanceVar);
        }
    }
}
```

打开命令行窗口，执行下面的命令编译本章代码：

```
go install jvmgo\ch06
```

命令执行完毕后，D:\go\workspace\bin 目录会出现 ch06.exe 文件。用 javac 编译 MyObject 类，然后用 ch06.exe 执行 MyObject 程序，结果如图 6-3 和图 6-4 所示。

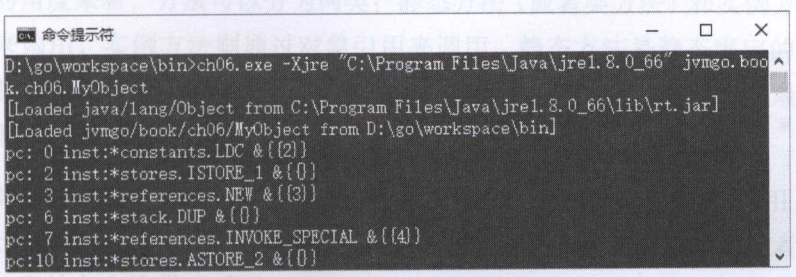


图 6-3 MyObject 程序执行结果（1）

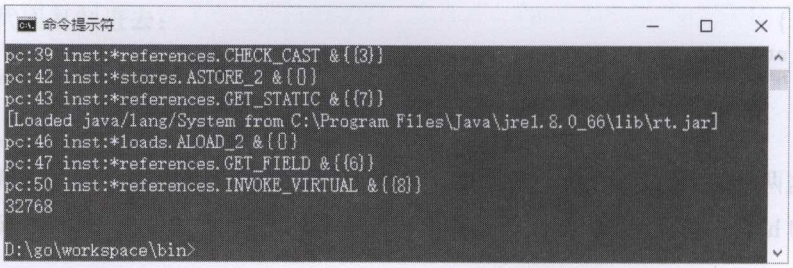
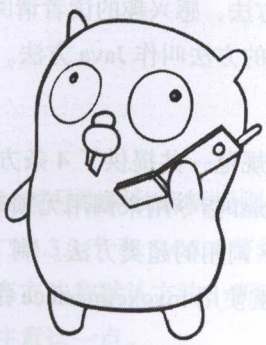


图 6-4 MyObject 程序执行结果（2）

6.8 本章小结

本章实现了方法区、运行时常量池、类和对象结构体、一个简单的类加载器，以及 ldc 和部分引用类指令。下一章将讨论方法调用和返回，到时就可以执行更加复杂的方法了。



第 7 章　方法调用和返回

第 4 章实现了 Java 虚拟机栈、帧等运行时数据区，为方法的执行打好了基础。第 5 章实现了一个简单的解释器和 150 多条指令，已经可以执行单个方法。第 6 章实现了方法区，为方法调用扫清了障碍。本章将实现方法调用和返回，在此基础上，还会讨论类和对象的初始化。

开始本章之前，还是先把目录结构准备好。复制 ch06 目录，改名为 ch07。修改 main.go 等文件，把 import 语句中的 ch06 全都替换成 ch07。本章对目录结构没有太大的调整。

7.1　方法调用概述

从调用的角度来看，方法可以分为两类：静态方法（或者类方法）和实例方法。静态方法通过类来调用，实例方法则通过对象引用来调用。静态方法是静态绑定的，也就是说，最终调用的是哪个方法在编译期就已经确定。实例方法则支持动态绑定，最终要调用哪个方法可能要推迟到运行期才能知道，本章将详细讨论这一点。

从实现的角度来看，方法可以分为三类：没有实现（也就是抽象方法）、用 Java 语言（或者 JVM 上的其他语言，如 Groovy 和 Scala 等）实现和用本地语言（如 C 或者 C++）实现。静态方法和抽象方法是互斥的。在 Java 8 之前，接口只能包含抽象方法。为了实现

Lambda 表达式，Java 8 放宽了这一限制，在接口中也可以定义静态方法和默认方法。本章不考虑接口的静态方法和默认方法，感兴趣的读者请阅读 Java 虚拟机规范相关章节。在本书中，我们把 Java 等语言实现的方法叫作 Java 方法。本章只讨论 Java 方法的调用，本地方法调用将在第 9 章中介绍。

在 Java 7 之前，Java 虚拟机规范一共提供了 4 条方法调用指令。其中 invokestatic 指令用来调用静态方法。invokespecial 指令用来调用无须动态绑定的实例方法，包括构造函数、私有方法和通过 super 关键字调用的超类方法。剩下的情况则属于动态绑定。如果是针对接口类型的引用调用方法，就使用 invokeinterface 指令，否则使用 invokevirtual 指令。本章将实现这 4 条指令。

为了更好地支持动态类型语言，Java 7 增加了一条方法调用指令 invokedynamic。本章不讨论这条指令，感兴趣的读者请阅读 Java 虚拟机规范相关章节。在深入讨论各条方法调用指令的细节之前，先简单了解 Java 虚拟机是如何调用方法的。

首先，方法调用指令需要 n+1 个操作数，其中第 1 个操作数是 uint16 索引，在字节码中紧跟在指令操作码的后面。通过这个索引，可以从当前类的运行时常量池中找到一个方法符号引用，解析这个符号引用就可以得到一个方法。注意，这个方法并不一定就是最终要调用的那个方法，所以可能还需要一个查找过程才能找到最终要调用的方法。剩下的 n 个操作数是要传递给被调用方法的参数，从操作数栈中弹出。将在 7.2 小节讨论方法符号引用的解析。

如果要执行的是 Java 方法（而非本地方法），下一步是给这个方法创建一个新的帧，并把它推到 Java 虚拟机栈顶。传递参数之后，新的方法就可以开始执行了。将在 7.3 小节讨论参数传递，本地方法调用则推迟到第 9 章再讨论。下面是一段伪代码，用于说明 Java 方法的调用过程。

```
func (self *INVOKE_XXX) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    methodRef := cp.GetConstant(self.Index).(*heap.MethodRef)
    resolved := resolveMethodRef(methodRef)
    checkResolvedMethod(resolved)
    toBeInvoked := findMethodToInvoke(methodRef)
    newFrame := frame.Thread().NewFrame(toBeInvoked)
    frame.Thread().PushFrame(newFrame)
    passArgs(frame, newFrame)
}
```

方法的最后一条指令是某个返回指令，这个指令负责把方法的返回值推入前一帧的操

作数栈顶，然后把当前帧从 Java 虚拟机栈中弹出。将在 7.4 小节讨论返回指令，在 7.5 小节讨论方法调用指令。

7.2　解析方法符号引用

非接口方法符号引用和接口方法符号引用的解析规则是不同的，因此本章分开讨论这两种符号引用。Java 虚拟机规范的 5.4.3.3 节和 5.4.3.4 节详细描述了这两种符号引用的解析规则。由于本书不讨论接口的静态方法和默认方法，所以在本小节中，主要参考 Java 虚拟机规范第 7 版编写代码，请读者注意这一点。

7.2.1　非接口方法符号引用

打开 ch07\rtda\heap\cp_methodref.go 文件，在其中实现 ResolvedMethod() 方法，代码如下：

```
func (self *MethodRef) ResolvedMethod() *Method {
    if self.method == nil {
        self.resolveMethodRef()
    }
    return self.method
}
```

如果还没有解析过符号引用，调用 resolveMethodRef() 方法进行解析，否则直接返回方法指针。resolveMethodRef() 方法的代码如下：

```
func (self *MethodRef) resolveMethodRef() {
    d := self.cp.class
    c := self.ResolvedClass()
    if c.IsInterface() {
        panic("java.lang.IncompatibleClassChangeError")
    }

    method := lookupMethod(c, self.name, self.descriptor)
    if method == nil {
        panic("java.lang.NoSuchMethodError")
    }
    if !method.isAccessibleTo(d) {
        panic("java.lang.IllegalAccessError")
    }

    self.method = method
}
```

如果类 D 想通过方法符号引用访问类 C 的某个方法，先要解析符号引用得到类 C。如果 C 是接口，则抛出 IncompatibleClassChangeError 异常，否则根据方法名和描述符查找方法。如果找不到对应的方法，则抛出 NoSuchMethodError 异常，否则检查类 D 是否有权限访问该方法。如果没有，则抛出 IllegalAccessError 异常。isAccessibleTo() 方法是在 ClassMember 结构体中定义的，在第 6 章就已经实现了。下面看一下 lookupMethod() 函数，其代码如下：

```
func lookupMethod(class *Class, name, descriptor string) *Method {
    method := LookupMethodInClass(class, name, descriptor)
    if method == nil {
        method = lookupMethodInInterfaces(class.interfaces, name, descriptor)
    }
    return method
}
```

先从 C 的继承层次中找，如果找不到，就去 C 的接口中找。LookupMethodInClass() 函数在很多地方都要用到，所以在 ch07\rtda\heap\method_lookup.go 文件中实现它，代码如下：

```
func LookupMethodInClass(class *Class, name, descriptor string) *Method {
    for c := class; c != nil; c = c.superClass {
        for _, method := range c.methods {
            if method.name == name && method.descriptor == descriptor {
                return method
            }
        }
    }
    return nil
}
```

lookupMethodInInterfaces() 函数也在 method_lookup.go 文件中，代码如下：

```
func lookupMethodInInterfaces(ifaces []*Class, name, descriptor string) *Method {
    for _, iface := range ifaces {
        for _, method := range iface.methods {
            if method.name == name && method.descriptor == descriptor {
                return method
            }
        }
        method := lookupMethodInInterfaces(iface.interfaces, name, descriptor)
        if method != nil {
            return method
        }
    }
```

```
    return nil
}
```

至此，非接口方法符号引用的解析就介绍完了，下面介绍接口方法符号引用如何解析。

7.2.2　接口方法符号引用

打开 ch07\rtda\heap\cp_interface_methodref.go 文件，在其中实现 ResolvedInterfaceMethod() 方法，代码如下：

```
func (self *InterfaceMethodRef) ResolvedInterfaceMethod() *Method {
    if self.method == nil {
        self. resolveInterfaceMethodRef()
    }
    return self.method
}
```

上面的代码和 ResolvedMethod() 方法大同小异，不多解释。下面来看 resolveInterface-MethodRef() 方法，代码如下：

```
func (self *InterfaceMethodRef) resolveInterfaceMethodRef() {
    d := self.cp.class
    c := self.ResolvedClass()
    if !c.IsInterface() {
        panic("java.lang.IncompatibleClassChangeError")
    }

    method := lookupInterfaceMethod(c, self.name, self.descriptor)
    if method == nil {
        panic("java.lang.NoSuchMethodError")
    }
    if !method.isAccessibleTo(d) {
        panic("java.lang.IllegalAccessError")
    }

    self.method = method
}
```

上面的代码和 resolveMethodRef () 方法也差不多，但有两处差别，已经用粗体显示。下面来看 lookupInterfaceMethod() 函数，代码如下：

```
func lookupInterfaceMethod(iface *Class, name, descriptor string) *Method {
    for _, method := range iface.methods {
        if method.name == name && method.descriptor == descriptor {
            return method
```

```
        }
    }
    return lookupMethodInInterfaces(iface.interfaces, name, descriptor)
}
```

如果能在接口中找到方法，就返回找到的方法，否则调用 lookupMethodInInterfaces() 函数在超接口中寻找。lookupMethodInInterfaces() 函数已经在前一小节介绍。至此，接口方法符号引用的解析也介绍完毕了，下面讨论如何给方法传递参数。

7.3 方法调用和参数传递

在定位到需要调用的方法之后，Java 虚拟机要给这个方法创建一个新的帧并把它推入 Java 虚拟机栈顶，然后传递参数。这个逻辑对于本章要实现的 4 条方法调用指令来说基本上相同，为了避免重复代码，在单独的文件中实现这个逻辑。在 ch07\instructions\base 目录下创建 method_invoke_logic.go 文件，在其中实现 InvokeMethod() 函数，代码如下：

```
func InvokeMethod(invokerFrame *rtda.Frame, method *heap.Method) {
    thread := invokerFrame.Thread()
    newFrame := thread.NewFrame(method)
    thread.PushFrame(newFrame)

    argSlotSlot := int(method.ArgSlotCount())
    if argSlotSlot > 0 {
        for i := argSlotSlot - 1; i >= 0; i-- {
            slot := invokerFrame.OperandStack().PopSlot()
            newFrame.LocalVars().SetSlot(uint(i), slot)
        }
    }
}
```

函数的前三行代码创建新的帧并推入 Java 虚拟机栈，剩下的代码传递参数。重点讨论参数传递。首先，要确定方法的参数在局部变量表中占用多少位置。注意，这个数量并不一定等于从 Java 代码中看到的参数个数，原因有两个。第一，long 和 double 类型的参数要占用两个位置。第二，对于实例方法，Java 编译器会在参数列表的前面添加一个参数，这个隐藏的参数就是 this 引用。假设实际的参数占据 *n* 个位置，依次把这 *n* 个变量从调用者的操作数栈中弹出，放进被调用方法的局部变量表中，参数传递就完成了。

注意，在代码中，并没有对 long 和 double 类型做特别处理。因为操作的是 Slot 结构体，所以这是没问题的。LocalVars 结构体的 SetSlot() 方法是新增的，代码如下：

```
func (self LocalVars) SetSlot(index uint, slot Slot) {
    self[index] = slot
}
```

如果忽略 long 和 double 类型参数，则静态方法的参数传递过程如图 7-1 所示。

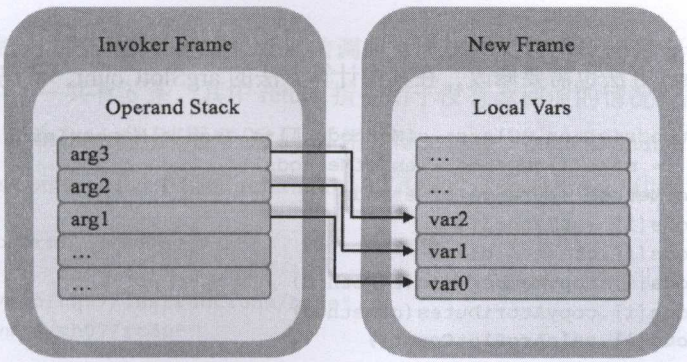


图 7-1　静态方法参数传递示意图

实例方法的参数传递过程如图 7-2 所示。

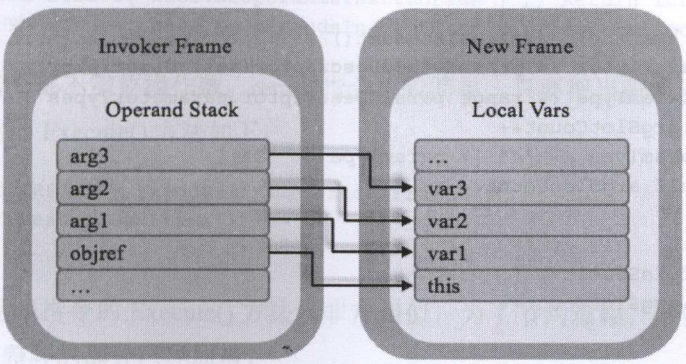


图 7-2　实例方法参数传递示意图

那么那个 ArgSlotCount() 方法返回了什么呢？打开 ch07\rtda\heap\method.go 文件，修改 Method 结构体，给它添加 argSlotCount 字段，代码如下：

```
type Method struct {
    ClassMember
    maxStack        uint
    maxLocals       uint
    code            []byte
    argSlotCount    uint
```

```
}
```

ArgSlotCount() 只是个 Getter 方法而已，代码如下：

```
func (self *Method) ArgSlotCount() uint {
    return self.argSlotCount
}
```

newMethods() 方法也需要修改，在其中计算方法的 argSlotCount，代码如下：

```
func newMethods(class *Class, cfMethods []*classfile.MemberInfo) []*Method {
    methods := make([]*Method, len(cfMethods))
    for i, cfMethod := range cfMethods {
        methods[i] = &Method{}
        methods[i].class = class
        methods[i].copyMemberInfo(cfMethod)
        methods[i].copyAttributes(cfMethod)
        methods[i].calcArgSlotCount()
    }
    return methods
}
```

下面是 calcArgSlotCount() 方法的代码。

```
func (self *Method) calcArgSlotCount() {
    parsedDescriptor := parseMethodDescriptor(self.descriptor)
    for _, paramType := range parsedDescriptor.parameterTypes {
        self.argSlotCount++
        if paramType == "J" || paramType == "D" {
            self.argSlotCount++
        }
    }
    if !self.IsStatic() {
        self.argSlotCount++
    }
}
```

parseMethodDescriptor() 函数分解方法描述符，返回一个 MethodDescriptor 结构体实例。这个结构体定义在 ch06\rtda\heap\method_descriptor.go 文件中，代码如下：

```
package heap

type MethodDescriptor struct {
    parameterTypes []string
    returnType     string
}
```

parseMethodDescriptor() 函数定义在 ch06\rtda\heap\method_descriptor_parser.go 文件

中，为了节约篇幅，这里就不详细介绍这个函数了，感兴趣的读者请阅读随书源代码。

7.4　返回指令

方法执行完毕之后，需要把结果返回给调用方，这一工作由返回指令完成。返回指令属于控制类指令，一共有 6 条。其中 return 指令用于没有返回值的情况，areturn、ireturn、lreturn、freturn 和 dreturn 分别用于返回引用、int、long、float 和 double 类型的值。在 ch07/instructions/control 目录下创建 return.go，在其中定义返回指令，代码如下：

```
package control

import "jvmgo/ch07/instructions/base"
import "jvmgo/ch07/rtda"

type RETURN struct{ base.NoOperandsInstruction } // Return void from method
type ARETURN struct{ base.NoOperandsInstruction } // Return reference from method
type DRETURN struct{ base.NoOperandsInstruction } // Return double from method
type FRETURN struct{ base.NoOperandsInstruction } // Return float from method
type IRETURN struct{ base.NoOperandsInstruction } // Return int from method
type LRETURN struct{ base.NoOperandsInstruction } // Return long from method
```

6 条返回指令都不需要操作数。return 指令比较简单，只要把当前帧从 Java 虚拟机栈中弹出即可，它的 Execute() 方法如下：

```
func (self *RETURN) Execute(frame *rtda.Frame) {
    frame.Thread().PopFrame()
}
```

其他 5 条返回指令的 Execute() 方法都非常相似，为了节约篇幅，下面只给出 ireturn 指令的代码（有差异的部分已经加粗）：

```
func (self *IRETURN) Execute(frame *rtda.Frame) {
    thread := frame.Thread()
    currentFrame := thread.PopFrame()
    invokerFrame := thread.TopFrame()
    retVal := currentFrame.OperandStack().PopInt()
    invokerFrame.OperandStack().PushInt(retVal)
}
```

Thread 结构体的 TopFrame() 方法和 CurrentFrame() 代码一样，这里用不同的名称主要是为了避免混淆。方法符号引用解析、参数传递、结果返回等都实现了，下面实现方法调用指令。

7.5 方法调用指令

与 7.2 节类似，由于本书不考虑接口的静态方法和默认方法，所以要实现的这 4 条指令并没有完全满足 Java 虚拟机规范第 8 版的规定，请读者留意这一点。下面从较简单的 invokestatic 指令开始。

7.5.1 invokestatic 指令

在 ch07\instructions\references 目录下创建 invokestatic.go 文件，在其中定义 invokestatic 指令，代码如下：

```
package references

import "jvmgo/ch07/instructions/base"
import "jvmgo/ch07/rtda"
import "jvmgo/ch07/rtda/class"

// Invoke a class (static) method
type INVOKE_STATIC struct{ base.Index16Instruction }
```

结构体定义比较简单，直接看 Execute() 方法，代码如下：

```
func (self *INVOKE_STATIC) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    methodRef := cp.GetConstant(self.Index).(*heap.MethodRef)
    resolvedMethod := methodRef.ResolvedMethod()
    if !resolved.IsStatic() {
        panic("java.lang.IncompatibleClassChangeError")
    }
    base.InvokeMethod(frame, resolvedMethod)
}
```

假定解析符号引用后得到方法 M。M 必须是静态方法，否则抛出 IncompatibleClassChangeError 异常。M 不能是类初始化方法。类初始化方法只能由 Java 虚拟机调用，不能使用 invokestatic 指令调用。这一规则由 class 文件验证器保证，这里不做检查。如果声明 M 的类还没有被初始化，则要先初始化该类。将在 7.8 小节讨论类的初始化。

对于 invokestatic 指令，M 就是最终要执行的方法，调用 InvokeMethod() 函数执行该方法。

7.5.2 invokespecial 指令

invokespecial 指令在第 6 章就定义好了，代码如下：

```
// Invoke instance method; special handling for superclass, private,
// and instance initialization method invocations
type INVOKE_SPECIAL struct{ base.Index16Instruction }
```

需要修改 Execute() 方法，代码稍微有些复杂，先看第一部分：

```
func (self *INVOKE_SPECIAL) Execute(frame *rtda.Frame) {
    currentClass := frame.Method().Class()
    cp := currentClass.ConstantPool()
    methodRef := cp.GetConstant(self.Index).(*heap.MethodRef)
    resolvedClass := methodRef.ResolvedClass()
    resolvedMethod := methodRef.ResolvedMethod()
```

先拿到当前类、当前常量池、方法符号引用，然后解析符号引用，拿到解析后的类和方法。继续看代码：

```
if resolvedMethod.Name() == "<init>" && resolvedMethod.Class() != resolvedClass {
    panic("java.lang.NoSuchMethodError")
}
if resolvedMethod.IsStatic() {
    panic("java.lang.IncompatibleClassChangeError")
}
```

假定从方法符号引用中解析出来的类是 C，方法是 M。如果 M 是构造函数，则声明 M 的类必须是 C，否则抛出 NoSuchMethodError 异常。如果 M 是静态方法，则抛出 IncompatibleClassChangeError 异常。继续看代码：

```
ref := frame.OperandStack().GetRefFromTop(resolvedMethod.ArgSlotCount())
if ref == nil {
    panic("java.lang.NullPointerException")
}
```

从操作数栈中弹出 this 引用，如果该引用是 null，抛出 NullPointerException 异常。注意，在传递参数之前，不能破坏操作数栈的状态。给 OperandStack 结构体添加一个 GetRefFromTop() 方法，该方法返回距离操作数栈顶 *n* 个单元格的引用变量。比如 GetRefFromTop(0) 返回操作数栈顶引用，GetRefFromTop(1) 返回从栈顶开始的倒数第二个引用，等等。GetRefFromTop() 方法的代码很简单，在后面给出。继续看 Execute() 方法：

```
if resolvedMethod.IsProtected() &&
    resolvedMethod.Class().IsSuperClassOf(currentClass) &&
    resolvedMethod.Class().GetPackageName() != currentClass.GetPackageName() &&
    ref.Class() != currentClass &&
    !ref.Class().IsSubClassOf(currentClass) {
```

```
        panic("java.lang.IllegalAccessError")
    }
```

上面的判断确保 protected 方法只能被声明该方法的类或子类调用。如果违反这一规定，则抛出 IllegalAccessError 异常。接着往下看：

```
    methodToBeInvoked := resolvedMethod
    if currentClass.IsSuper() &&
        resolvedClass.IsSuperClassOf(currentClass) &&
        resolvedMethod.Name() != "<init>" {

        methodToBeInvoked = heap.LookupMethodInClass(currentClass.SuperClass(),
            methodRef.Name(), methodRef.Descriptor())
    }
```

上面这段代码比较难懂，把它翻译成更容易理解的语言：如果调用的中超类中的函数，但不是构造函数，且当前类的 ACC_SUPER 标志被设置，需要一个额外的过程查找最终要调用的方法；否则前面从方法符号引用中解析出来的方法就是要调用的方法。

```
        if methodToBeInvoked == nil || methodToBeInvoked.IsAbstract() {
            panic("java.lang.AbstractMethodError")
        }

        base.InvokeMethod(frame, methodtoBeInvoked)
    }
```

如果查找过程失败，或者找到的方法是抽象的，抛出 AbstractMethodError 异常。最后，如果一切正常，就调用方法。这里之所以这么复杂，是因为调用超类的（非构造函数）方法需要特别处理。限于篇幅，这里就不深入讨论了，读者可以阅读 Java 虚拟机规范，了解类的 ACC_SUPER 访问标志的用法。

OperandStack 结构体 GetRefFromTop() 方法的代码如下：

```
func (self *OperandStack) GetRefFromTop(n uint) *heap.Object {
    return self.slots[self.size-1-n].ref
}
```

7.5.3 invokevirtual 指令

invokevirtual 指令也已经在第 6 章定义好了，代码如下：

```
// Invoke instance method; dispatch based on class
type INVOKE_VIRTUAL struct{ base.Index16Instruction }
```

下面修改 Execute() 方法，第一部分代码如下：

```
func (self *INVOKE_VIRTUAL) Execute(frame *rtda.Frame) {
    currentClass := frame.Method().Class()
    cp := currentClass.ConstantPool()
    methodRef := cp.GetConstant(self.Index).(*heap.MethodRef)
    resolvedMethod := methodRef.ResolvedMethod()
    if resolvedMethod.IsStatic() {
        panic("java.lang.IncompatibleClassChangeError")
    }

    ref := frame.OperandStack().GetRefFromTop(resolvedMethod.ArgSlotCount() - 1)
    if ref == nil {
        // hack System.out.println()
        panic("java.lang.NullPointerException")
    }

    if resolvedMethod.IsProtected() &&
        resolvedMethod.Class().IsSuperClassOf(currentClass) &&
        resolvedMethod.Class().GetPackageName() != currentClass.GetPackageName() &&
        ref.Class() != currentClass &&
        !ref.Class().IsSubClassOf(currentClass) {

        panic("java.lang.IllegalAccessError")
    }
```

这部分代码和 invokespecial 指令基本上差不多，也不多解释了。下面是剩下的代码。

```
    methodToBeInvoked := heap.LookupMethodInClass(ref.Class(),
        methodRef.Name(), methodRef.Descriptor())
    if methodToBeInvoked == nil || methodToBeInvoked.IsAbstract() {
        panic("java.lang.AbstractMethodError")
    }

    base.InvokeMethod(frame, methodToBeInvoked)
}
```

从对象的类中查找真正要调用的方法。如果找不到方法，或者找到的是抽象方法，则需要抛出 AbstractMethodError 异常，否则一切正常，调用方法。注意，仍然要用 hack 的方式调用 System.out.println() 方法，代码如下：

```
func (self *INVOKE_VIRTUAL) Execute(frame *rtda.Frame) {
    ... // 其他代码
    ref := frame.OperandStack().GetRefFromTop(resolvedMethod.ArgSlotCount() - 1)
    if ref == nil {
        // hack!
        if methodRef.Name() == "println" {
```

```
            _println(frame.OperandStack(), methodRef.Descriptor())
            return
        }
        panic("java.lang.NullPointerException")
    }
    ... // 其他代码
}
```

_println() 函数如下：

```
func _println(stack *rtda.OperandStack, descriptor string) {
    switch descriptor {
    case "(Z)V": fmt.Printf("%v\n", stack.PopInt() != 0)
    case "(C)V": fmt.Printf("%c\n", stack.PopInt())
    case "(B)V": fmt.Printf("%v\n", stack.PopInt())
    case "(S)V": fmt.Printf("%v\n", stack.PopInt())
    case "(I)V": fmt.Printf("%v\n", stack.PopInt())
    case "(F)V": fmt.Printf("%v\n", stack.PopFloat())
    case "(J)V": fmt.Printf("%v\n", stack.PopLong())
    case "(D)V": fmt.Printf("%v\n", stack.PopDouble())
    default: panic("println: " + descriptor)
    }
    stack.PopRef()
}
```

7.5.4 invokeinterface 指令

在 ch07\instructions\references 目录下创建 invokeinterface.go 文件，在其中定义 invokeinterface 指令，代码如下：

```
package references

import "jvmgo/ch07/instructions/base"
import "jvmgo/ch07/rtda"
import "jvmgo/ch07/rtda/heap"

// Invoke interface method
type INVOKE_INTERFACE struct {
    index uint
    // count uint8
    // zero uint8
}
```

注意，和其他三条方法调用指令略有不同，在字节码中，invokeinterface 指令的操作码后面跟着 4 字节而非 2 字节。前两字节的含义和其他指令相同，是个 uint16 运行时常量池索引。第 3 字节的值是给方法传递参数需要的 slot 数，其含义和给 Method 结构体定义

的 argSlotCount 字段相同。正如我们所知，这个数是可以根据方法描述符计算出来的，它的存在仅仅是因为历史原因。第 4 字节是留给 Oracle 的某些 Java 虚拟机实现用的，它的值必须是 0。该字节的存在是为了保证 Java 虚拟机可以向后兼容。

invokeinterface 指令的 FetchOperands() 方法如下：

```
func (self *INVOKE_INTERFACE) FetchOperands(reader *base.BytecodeReader) {
    self.index = uint(reader.ReadUint16())
    reader.ReadUint8() // count
    reader.ReadUint8() // must be 0
}
```

下面看 Execute() 方法，第一部分代码如下：

```
func (self *INVOKE_INTERFACE) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    methodRef := cp.GetConstant(self.index).(*heap.InterfaceMethodRef)
    resolvedMethod := methodRef.ResolvedInterfaceMethod()
    if resolvedMethod.IsStatic() || resolvedMethod.IsPrivate() {
        panic("java.lang.IncompatibleClassChangeError")
    }
```

先从运行时常量池中拿到并解析接口方法符号引用，如果解析后的方法是静态方法或私有方法，则抛出 IncompatibleClassChangeError 异常。继续看代码：

```
    ref := frame.OperandStack().GetRefFromTop(resolvedMethod.ArgSlotCount() - 1)
    if ref == nil {
        panic("java.lang.NullPointerException")
    }
    if !ref.Class().IsImplements(methodRef.ResolvedClass()) {
        panic("java.lang.IncompatibleClassChangeError")
    }
```

从操作数栈中弹出 this 引用，如果引用是 null，则抛出 NullPointerException 异常。如果引用所指对象的类没有实现解析出来的接口，则抛出 IncompatibleClassChangeError 异常。继续看代码：

```
    methodToBeInvoked := heap.LookupMethodInClass(ref.Class(),
        methodRef.Name(), methodRef.Descriptor())
    if methodToBeInvoked == nil || methodToBeInvoked.IsAbstract() {
        panic("java.lang.AbstractMethodError")
    }
    if !methodToBeInvoked.IsPublic() {
        panic("java.lang.IllegalAccessError")
    }
```

查找最终要调用的方法。如果找不到，或者找到的方法是抽象的，则抛出 AbstractMethodError 异常。如果找到的方法不是 public，则抛出 IllegalAccessError 异常，否则，一切正常，调用方法：

```
    base.InvokeMethod(frame, methodToBeInvoked)
}
```

至此，4 条方法调用指令都实现完毕了。再总结一下这 4 条指令的用途。invokestatic 指令调用静态方法，很好理解。invokespecial 指令也比较好理解。首先，因为私有方法和构造函数不需要动态绑定，所以 invokespecial 指令可以加快方法调用速度。其次，使用 super 关键字调用超类中的方法不能使用 invokevirtual 指令，否则会陷入无限循环。

那么为什么要单独定义 invokeinterface 指令呢？统一使用 invokevirtual 指令不行吗？答案是，可以，但是可能会影响效率。这两条指令的区别在于：当 Java 虚拟机通过 invokevirtual 调用方法时，this 引用指向某个类（或其子类）的实例。因为类的继承层次是固定的，所以虚拟机可以使用一种叫作 vtable（Virtual Method Table）的技术加速方法查找。但是当通过 invokeinterface 指令调用接口方法时，因为 this 引用可以指向任何实现了该接口的类的实例，所以无法使用 vtable 技术。

由于篇幅限制，这里就不深入讨论 vtable 技术了。感兴趣的读者可以阅读相关资料，或者改进我们的代码，给 invokevirtual 指令增加 vtable 优化。

4 条方法调用指令和 6 条返回指令都准备好了，还需要修改 ch07\instructions\factory.go 文件，在其中增加这些指令的 case 语句。鉴于改动比较简单，这里就不给出代码了。

7.6 改进解释器

我们的解释器目前只能执行单个方法，现在就扩展它，让它支持方法调用。打开 ch07\interpreter.go 文件，修改 interpret() 方法，代码如下：

```
func interpret(method *heap.Method, logInst bool) {
    thread := rtda.NewThread()
    frame := thread.NewFrame(method)
    thread.PushFrame(frame)

    defer catchErr(thread)
    loop(thread, logInst)
}
```

logInst 参数控制是否把指令执行信息打印到控制台。更重要的变化在 loop() 函数中，代码如下所示：

```
func loop(thread *rtda.Thread, logInst bool) {
    reader := &base.BytecodeReader{}
    for {
        frame := thread.CurrentFrame()
        pc := frame.NextPC()
        thread.SetPC(pc)

        // decode
        reader.Reset(frame.Method().Code(), pc)
        opcode := reader.ReadUint8()
        inst := instructions.NewInstruction(opcode)
        inst.FetchOperands(reader)
        frame.SetNextPC(reader.PC())

        if (logInst) {
            logInstruction(frame, inst)
        }

        // execute
        inst.Execute(frame)
        if thread.IsStackEmpty() {
            break
        }
    }
}
```

在每次循环开始，先拿到当前帧，然后根据 pc 从当前方法中解码出一条指令。指令执行完毕之后，判断 Java 虚拟机栈中是否还有帧。如果没有则退出循环；否则继续。Thread 结构体的 IsStackEmpty() 方法是新增加的，代码在 ch07\rtda\thread.go 中，如下所示：

```
func (self *Thread) IsStackEmpty() bool {
    return self.stack.isEmpty()
}
```

它只是调用了 Stack 结构体的 isEmpty() 方法，代码在 ch07\rtda\jvm_stack.go 中，如下所示：

```
func (self *Stack) isEmpty() bool {
    return self._top == nil
}
```

回到 interpreter.go，如果解释器在执行期间出现了问题，catchErr() 函数会打印出错信息，代码如下：

```
func catchErr(thread *rtda.Thread) {
    if r := recover(); r != nil {
        logFrames(thread)
        panic(r)
    }
}
```

logFrames() 函数打印 Java 虚拟机栈信息，代码如下：

```
func logFrames(thread *rtda.Thread) {
    for !thread.IsStackEmpty() {
        frame := thread.PopFrame()
        method := frame.Method()
        className := method.Class().Name()
        fmt.Printf(">> pc:%4d %v.%v%v \n",
            frame.NextPC(), className, method.Name(), method.Descriptor())
    }
}
```

logInstruction() 函数在方法执行过程中打印指令信息，代码如下：

```
func logInstruction(frame *rtda.Frame, inst base.Instruction) {
    method := frame.Method()
    className := method.Class().Name()
    methodName := method.Name()
    pc := frame.Thread().PC()
    fmt.Printf("%v.%v() #%2d %T %v\n", className, methodName, pc, inst, inst)
}
```

解释器改造完毕，下面测试方法调用。

7.7 测试方法调用

先改造命令行工具，给它增加两个选项。java 命令提供了 -verbose:class（简写为 -verbose）选项，可以控制是否把类加载信息输出到控制台。也增加这样一个选项，另外参照这个选项增加一个 -verbose:inst 选项，用来控制是否把指令执行信息输出到控制台。

打开 ch07\cmd.go 文件，修改 Cmd 结构体如下：

```
type Cmd struct {
    helpFlag            bool
    versionFlag         bool
    verboseClassFlag    bool
    verboseInstFlag     bool
    cpOption            string
```

```
    XjreOption      string
    class           string
    args            []string
}
```

parseCmd() 函数也需要修改，改动比较简单，这里就不给出代码了。下面修改 ch07\main.go 文件，其他地方不变，只需要修改 startJVM() 函数，代码如下：

```
func startJVM(cmd *Cmd) {
    cp := classpath.Parse(cmd.XjreOption, cmd.cpOption)
    classLoader := heap.NewClassLoader(cp, cmd.verboseClassFlag)

    className := strings.Replace(cmd.class, ".", "/", -1)
    mainClass := classLoader.LoadClass(className)
    mainMethod := mainClass.GetMainMethod()
    if mainMethod != nil {
        interpret(mainMethod, cmd.verboseInstFlag)
    } else {
        fmt.Printf("Main method not found in class %s\n", cmd.class)
    }
}
```

然后修改 ch07\rtda\heap\class_loader.go 文件，给 ClassLoader 结构体添加 verboseFlag 字段，代码如下：

```
type ClassLoader struct {
    cp          *classpath.Classpath
    verboseFlag bool
    classMap    map[string]*Class
}
```

NewClassLoader() 函数要相应修改，改动如下：

```
func NewClassLoader(cp *classpath.Classpath, verboseFlag bool) *ClassLoader {
    return &ClassLoader{
        cp:          cp,
        verboseFlag: verboseFlag,
        classMap:    make(map[string]*Class),
    }
}
```

loadNonArrayClass() 函数也要修改，改动如下：

```
func (self *ClassLoader) loadNonArrayClass(name string) *Class {
    data, entry := self.readClass(name)
    class := self.defineClass(data)
    link(class)
```

```
    if self.verboseFlag {
        fmt.Printf("[Loaded %s from %s]\n", name, entry)
    }
    return class
}
```

一切都准备就绪，打开命令行窗口，执行下面的命令编译本章代码：

```
go install jvmgo\ch07
```

命令执行完毕后，在 D:\go\workspace\bin 目录下出现 ch07.exe 文件。下面这个类演示了各种情况下，4 种方法调用命令的使用。

```
package jvmgo.book.ch07;

public class InvokeDemo implements Runnable {
    public static void main(String[] args) {
        new InvokeDemo().test();
    }
    public void test() {
        InvokeDemo.staticMethod();                  // invokestatic
        InvokeDemo demo = new InvokeDemo();         // invokespecial
        demo.instanceMethod();                      // invokespecial
        super.equals(null);                         // invokespecial
        this.run();                                 // invokevirtual
        ((Runnable) demo).run();                    // invokeinterface
    }
    public static void staticMethod() {}
    private void instanceMethod() {}
    @Override public void run() {}
}
```

用 javac 编译 InvokeDemo 类，然后用 ch07.exe 执行 InvokeDemo 程序，可以看到程序正常执行（没有任何输出），如图 7-3 所示。

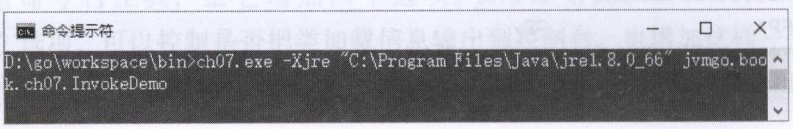


图 7-3 InvokeDemo 执行结果

InvokeDemo 只是演示，下面看一个稍微复杂一些的例子。

```
package jvmgo.book.ch07;

public class FibonacciTest {
```

```
    public static void main(String[] args) {
        long x = fibonacci(30);
        System.out.println(x);
    }
    private static long fibonacci(long n) {
        if (n <= 1) { return n; }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

FibonacciTest 类演示了斐波那契数列的计算，用 javac 编译它，然后用 ch07.exe 执行，结果如图 7-4 所示。

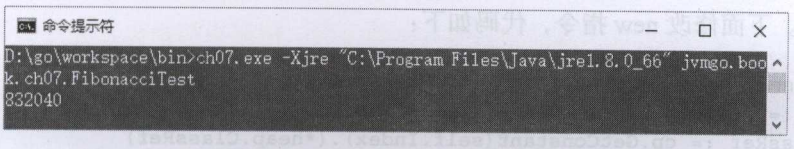


图 7-4　FibonacciTest 执行结果

几秒钟停顿之后，控制台上打印出了 832040。我们的 Java 虚拟机终于可以执行复杂计算了。方法调用指令就测试到这里，下面在本章的最后，讨论类的初始化。

7.8　类初始化

第 6 章实现了一个简化版的类加载器，可以把类加载到方法区中。但是因为当时还没有实现方法调用，所以没有办法初始化类。现在可以把这个逻辑补上了。我们已经知道，类初始化就是执行类的初始化方法（<clinit>）。类的初始化在下列情况下触发：

- 执行 new 指令创建类实例，但类还没有被初始化。
- 执行 putstatic、getstatic 指令存取类的静态变量，但声明该字段的类还没有被初始化。
- 执行 invokestatic 调用类的静态方法，但声明该方法的类还没有被初始化。
- 当初始化一个类时，如果类的超类还没有被初始化，要先初始化类的超类。
- 执行某些反射操作时。

为了判断类是否已经初始化，需要给 Class 结构体添加一个字段：

```
type Class struct {
    ... // 其他字段
    initStarted bool
}
```

类的初始化其实分为几个阶段，但由于我们的类加载器还不够完善，所以先使用一个简单的布尔状态就足够了。initStarted 字段表示类的 <clinit> 方法是否已经开始执行。接下来给 Class 结构体添加两个方法，代码如下：

```
func (self *Class) InitStarted() bool {
   return self.initStarted
}
func (self *Class) StartInit() {
   self.initStarted = true
}
```

InitStarted() 是 Getter 方法，返回 initStarted 字段值。StartInit() 方法把 initStarted 字段设置成 true。下面修改 new 指令，代码如下：

```
func (self *NEW) Execute(frame *rtda.Frame) {
   cp := frame.Method().Class().ConstantPool()
   classRef := cp.GetConstant(self.Index).(*heap.ClassRef)
   class := classRef.ResolvedClass()
   if !class.InitStarted() {
      frame.RevertNextPC()
      base.InitClass(frame.Thread(), class)
      return
   }
   ... // 其他代码
}
```

putstatic 和 getstatic 指令改动类似，以 putstatic 指令为例，代码如下：

```
func (self *PUT_STATIC) Execute(frame *rtda.Frame) {
   ... // 其他代码
   field := fieldRef.ResolvedField()
   class := field.Class()
   if !class.InitStarted() {
      frame.RevertNextPC()
      base.InitClass(frame.Thread(), class)
      return
   }
   ... // 其他代码
}
```

invokestatic 指令也需要修改，改动如下：

```
func (self *INVOKE_STATIC) Execute(frame *rtda.Frame) {
   ... // 其他代码
   class := resolvedMethod.Class()
   if !class.InitStarted() {
```

```
            frame.RevertNextPC()
            base.InitClass(frame.Thread(), class)
            return
        }
        base.InvokeMethod(frame, resolvedMethod)
    }
```

4 条指令都修改完毕了，但是新增加的代码做了些什么？先判断类的初始化是否已经开始，如果还没有，则需要调用类的初始化方法，并终止指令执行。但是由于此时指令已经执行到了一半，也就是说当前帧的 nextPC 字段已经指向下一条指令，所以需要修改 nextPC，让它重新指向当前指令。Frame 结构体的 RevertNextPC() 方法做了这样的操作，代码如下：

```
func (self *Frame) RevertNextPC() {
    self.nextPC = self.thread.pc
}
```

nextPC 调整好之后，下一步查找并调用类的初始化方法。这个逻辑是通用的，在 ch07\instructions\base\class_init_logic.go 文件中实现它，代码如下：

```
func InitClass(thread *rtda.Thread, class *heap.Class) {
    class.StartInit()
    scheduleClinit(thread, class)
    initSuperClass(thread, class)
}
```

InitClass() 函数先调用 StartInit() 方法把类的 initStarted 状态设置成 true 以免进入死循环，然后调用 scheduleClinit() 函数准备执行类的初始化方法，代码如下：

```
func scheduleClinit(thread *rtda.Thread, class *heap.Class) {
    clinit := class.GetClinitMethod()
    if clinit != nil {
        // exec <clinit>
        newFrame := thread.NewFrame(clinit)
        thread.PushFrame(newFrame)
    }
}
```

类初始化方法没有参数，所以不需要传递参数。Class 结构体的 GetClinitMethod() 方法如下：

```
func (self *Class) GetClinitMethod() *Method {
    return self.getStaticMethod("<clinit>", "()V")
}
```

注意，这里有意使用了 scheduleClinit 这个函数名而非 invokeClinit，因为有可能要先执行超类的初始化方法，如函数 initSuperClass() 所示。

```
func initSuperClass(thread *rtda.Thread, class *heap.Class) {
    if !class.IsInterface() {
        superClass := class.SuperClass()
        if superClass != nil && !superClass.InitStarted() {
            InitClass(thread, superClass)
        }
    }
}
```

如果超类的初始化还没有开始，就递归调用 InitClass() 函数执行超类的初始化方法，这样可以保证超类的初始化方法对应的帧在子类上面，使超类初始化方法先于子类执行。

类的初始化逻辑写完了，由于篇幅限制，这里就不进行测试了。读者可以参考随书 Java 示例代码，或者自行编写 Java 程序进行测试。不过在进行测试之前，还需要增加一个小小的 hack。由于目前还不支持本地方法调用，而 Java 类库中的很多类都要注册本地方法，比如 Object 类就有一个 registerNatives() 本地方法，用于注册其他方法，代码如下：

```
// java.lang.Object
public class Object {
    private static native void registerNatives();
    static {
        registerNatives();
    }
    ... // 其他代码
}
```

由于 Object 类是其他所有类的超类，所以这会导致 Java 虚拟机崩溃。解决办法是修改 InvokeMethod() 函数（代码在 ch07\instructions\base\method_invoke_logic.go 文件中），让它跳过所有 registerNatives() 方法，改动如下：

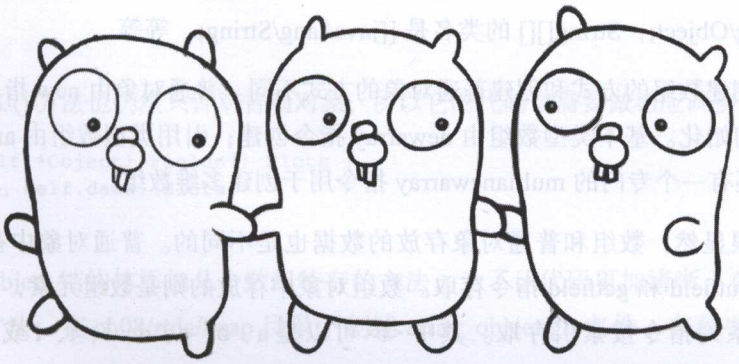
```
package base

import "fmt"
import "jvmgo/ch07/rtda"
import "jvmgo/ch07/rtda/heap"

func InvokeMethod(invokerFrame *rtda.Frame, method *heap.Method) {
    ... // 前面的代码不变，下面是 hack！
    if method.IsNative() {
        if method.Name() == "registerNatives" {
            thread.PopFrame()
```

```
        } else {
            panic(fmt.Sprintf("native method: %v.%v%v\n",
                method.Class().Name(), method.Name(), method.Descriptor()))
        }
    }
}
```

如果遇到其他本地方法，直接调用 panic() 函数终止程序执行即可。将在第 9 章讨论本地方法调用。

7.9　本章小结

本章讨论了方法调用和返回，并且实现了类初始化逻辑。如果说在前面几章里，我们的 Java 虚拟机还是个小 baby 只会爬的话，到了本章结尾，它已经可以满地跑了。下一章将讨论数组和字符串，届时，我们的小 baby 就有更多的玩具可以玩耍了。



第 8 章　数组和字符串

在大部分编程语言中，数组和字符串都是最基本的数据类型。Java 虚拟机直接支持数组，对字符串的支持则由 java.lang.String 和相关的类提供。本章分为两部分，前半部分讨论数组和数组相关指令，后半部分讨论字符串。

在本章的讨论中，**数组**一般指**数组对象**；在不至于引起混淆的情况下，也可能指**数组类**；在其他情况下，会明确指出是**数组类**还是**数组对象**。如果数组的元素是基本类型，就把它叫作**基本类型数组**，否则，数组的元素是引用类型，把它叫作**引用类型数组**。基本类型数组肯定都是**一维数组**，如果引用类型数组的元素也是数组，那么它就是**多维数组**。

开始学习本章之前，还是先把目录结构准备好。复制 ch07 目录，改名为 ch08。修改 main.go 等文件，把 import 语句中的 ch07 全都替换成 ch08。本章对目录结构没有太大的调整。

8.1　数组概述

数组在 Java 虚拟机中是个比较特殊的概念。为什么这么说呢？有下面几个原因：

首先，数组类和普通的类是不同的。普通的类从 class 文件中加载，但是数组类由 Java 虚拟机在运行时生成。数组的类名是左方括号（[）+ 数组元素的类型描述符；数组的

类型描述符就是类名本身。例如，int[] 的类名是 [I，int[][] 的类名是 [[I，Object[] 的类名是 [Ljava/lang/Object;，String[][] 的类名是 [[java/lang/String;，等等。

其次，创建数组的方式和创建普通对象的方式不同。普通对象由 new 指令创建，然后由构造函数初始化。基本类型数组由 newarray 指令创建；引用类型数组由 anewarray 指令创建；另外还有一个专门的 multianewarray 指令用于创建多维数组。

最后，很显然，数组和普通对象存放的数据也是不同的。普通对象中存放的是实例变量，通过 putfield 和 getfield 指令存取。数组对象中存放的则是数组元素，通过 <t>aload 和 <t>astore 系列指令按索引存取。其中 <t> 可以是 a、b、c、d、f、i、l 或者 s，分别用于存取引用、byte、char、double、float、int、long 或 short 类型的数组。另外，还有一个 arraylength 指令，用于获取数组长度。

Java 虚拟机规范给了实现者充分的自由来实现数组，下面就动手吧！

8.2 数组实现

将在类和对象的基础上实现数组类和数组对象，先来看数组对象。

8.2.1 数组对象

和普通对象一样，数组也是分配在堆中的，通过引用来使用。所以需要改造 Object 结构体，让它既可以表示普通的对象，也可以表示数组。打开 ch08\rtda\heap\object.go，修改 Object 结构体，改动如下：

```
type Object struct {
    class    *Class
    data     interface{}
}
```

把 fields 字段改为 data，类型也从 Slots 变成了 interface{}。Go 语言的 interface{} 类型很像 C 语言中的 void*，该类型的变量可以容纳任何类型的值。对于普通对象来说，data 字段中存放的仍然还是 Slots 变量。但是对于数组，可以在其中放各种类型的数组，详见下文。newObject() 用来创建普通对象，因此需要做相应的调整，改动如下：

```
func newObject(class *Class) *Object {
    return &Object{
        class: class,
```

```
        data: newSlots(class.instanceSlotCount),
    }
}
```

因为 Fields() 方法也仍然只针对普通对象，所以它的代码也需要做相应调整，如下所示：

```
func (self *Object) Fields() Slots {
    return self.data.(Slots)
}
```

需要给 Object 结构体添加几个数组特有的方法，为了让代码更加清晰，在单独的文件中定义这些方法。在 ch08/rtda/heap 目录下创建 array_object.go 文件，在其中实现 8 个方法，代码如下：

```
package heap

func (self *Object) Bytes() []int8 { return self.data.([]int8) }
func (self *Object) Shorts() []int16 { return self.data.([]int16) }
func (self *Object) Ints() []int32 { return self.data.([]int32) }
func (self *Object) Longs() []int64 { return self.data.([]int64) }
func (self *Object) Chars() []uint16 { return self.data.([]uint16) }
func (self *Object) Floats() []float32 { return self.data.([]float32) }
func (self *Object) Doubles() []float64 { return self.data.([]float64) }
func (self *Object) Refs() []*Object { return self.data.([]*Object) }
```

上面这 8 个方法分别针对引用类型数组和 7 种基本类型数组返回具体的数组数据。继续编辑 array_object.go 文件，在其中添加 ArrayLength() 方法，代码如下：

```
func (self *Object) ArrayLength() int32 {
    switch self.fields.(type) {
    case []int8: return int32(len(self.data.([]int8)))
    case []int16: return int32(len(self.data.([]int16)))
    case []int32: return int32(len(self.data.([]int32)))
    case []int64: return int32(len(self.data.([]int64)))
    case []uint16: return int32(len(self.data.([]uint16)))
    case []float32: return int32(len(self.data.([]float32)))
    case []float64: return int32(len(self.data.([]float64)))
    case []*Object: return int32(len(self.data.([]*Object)))
    default: panic("Not array!")
    }
}
```

读者也许会好奇，为什么返回数组数据的方法有 8 个，但却只有一个统一的 ArrayLength() 方法呢？答案是，这些方法主要是供 <t>aload、<t>astore 和 arraylength 指令使用的。<t>aload 和 <t>astore 系列指令各有 8 条，所以针对每种类型都提供一个方法，返

回相应的数组数据。因为 arraylength 指令只有一条，所以 ArrayLength() 方法需要自己判断数组类型。

那么为什么没有实现 Booleans() 方法呢？因为将使用 []int8 来表示布尔数组，所以只需要 Bytes() 方法即可。心急的读者可以先跳到 8.3 小节，看看数组相关指令是如何实现的。下面实现数组类。

8.2.2 数组类

不需要修改 Class 结构体，只给它添加几个数组特有的方法即可。为了强调这些方法只针对数组类，同时也避免 class.go 文件变得过长，把这些方法定义在新的文件中。

在 ch08/rtda/heap 目录下创建 array_class.go 文件，在其中定义 NewArray() 方法，代码如下：

```
func (self *Class) NewArray(count uint) *Object {
    if !self.IsArray() {
        panic("Not array class: " + self.name)
    }
    switch self.Name() {
    case "[Z": return &Object{self, make([]int8, count)}
    case "[B": return &Object{self, make([]int8, count)}
    case "[C": return &Object{self, make([]uint16, count)}
    case "[S": return &Object{self, make([]int16, count)}
    case "[I": return &Object{self, make([]int32, count)}
    case "[J": return &Object{self, make([]int64, count)}
    case "[F": return &Object{self, make([]float32, count)}
    case "[D": return &Object{self, make([]float64, count)}
    default: return &Object{self, make([]*Object, count)}
    }
}
```

NewArray() 方法专门用来创建数组对象。如果类并不是数组类，就调用 panic() 函数终止程序执行，否则根据数组类型创建数组对象。注意：布尔数组是使用字节数组来表示的。继续编辑 array_class.go，在其中定义 IsArray() 方法，代码如下：

```
func (self *Class) IsArray() bool {
    return self.name[0] == '['
}
```

还会在 array_class.go 文件中实现其他几个方法，等用到时再介绍。下面修改类加载器，让它可以加载数组类。

8.2.3　加载数组类

打开 ch08\rtda\heap\class_loader.go 文件，修改 LoadClass() 方法，改动如下：

```
func (self *ClassLoader) LoadClass(name string) *Class {
    if class, ok := self.classMap[name]; ok {
        return class // 已经加载
    }
    if name[0] == '[' {
        return self.loadArrayClass(name)
    }
    return self.loadNonArrayClass(name)
}
```

这里增加了类型判断，如果要加载的类是数组类，则调用新的 loadArrayClass() 方法，否则还按照原来的逻辑。loadArrayClass() 方法需要生成一个 Class 结构体实例，代码如下：

```
func (self *ClassLoader) loadArrayClass(name string) *Class {
    class := &Class{
        accessFlags:       ACC_PUBLIC, // todo
        name:              name,
        loader:            self,
        initStarted:       true,
        superClass:        self.LoadClass("java/lang/Object"),
        interfaces:        []*Class{
            self.LoadClass("java/lang/Cloneable"),
            self.LoadClass("java/io/Serializable"),
        },
    }
    self.classMap[name] = class
    return class
}
```

前面三个字段的值比较好理解，不多解释。因为数组类不需要初始化，所以把 initStarted 字段设置成 true。数组类的超类是 java.lang.Object，并且实现了 java.lang.Cloneable 和 java.io.Serializable 接口。类加载器改造完毕，下面来实现数组相关指令。

8.3　数组相关指令

本节要实现 20 条指令，其中 newarray、anewarray、multianewarray 和 arraylength 指令属于引用类指令；<t>aload 和 <t>astore 系列指令各有 8 条，分别属于加载类和存储类指令。下面的 Java 程序演示了部分数组相关指令的用处。

```
public class ArrayDemo {
    public static void main(String[] args) {
        int[] a1 = new int[10];          // newarray
        String[] a2 = new String[10];    // anewarray
        int[][] a3 = new int[10][10];    // multianewarray
        int x = a1.length;               // arraylength
        a1[0] = 100;                     // iastore
        int y = a1[0];                   // iaload
        a2[0] = "abc";                   // aastore
        String s = a2[0];                // aaload
    }
}
```

下面从 newarray 指令开始。

8.3.1 newarray 指令

newarray 指令用来创建基本类型数组，包括 boolean[]、byte[]、char[]、short[]、int[]、long[]、float[] 和 double[] 8 种。在 ch08\instructions\references 目录下创建 newarray.go，在其中定义 newarray 指令，代码如下：

```
package references

import "jvmgo/ch08/instructions/base"
import "jvmgo/ch08/rtda"
import "jvmgo/ch08/rtda/heap"

const (...) // atype 常量

// Create new array of primitive
type NEW_ARRAY struct {
    atype uint8
}
```

newarray 指令需要两个操作数。第一个操作数是一个 uint8 整数，在字节码中紧跟在指令操作码后面，表示要创建哪种类型的数组。Java 虚拟机规范把这个操作数叫作 atype，并且规定了它的有效值。把这些值定义为常量，代码如下：

```
const (
    AT_BOOLEAN  = 4
    AT_CHAR     = 5
    AT_FLOAT    = 6
    AT_DOUBLE   = 7
    AT_BYTE     = 8
    AT_SHORT    = 9
```

```
    AT_INT          = 10
    AT_LONG         = 11
)
```

FetchOperands() 方法读取 atype 的值，代码如下：

```
func (self *NEW_ARRAY) FetchOperands(reader *base.BytecodeReader) {
    self.atype = reader.ReadUint8()
}
```

newarray 指令的第二个操作数是 count，从操作数栈中弹出，表示数组长度。Execute() 方法根据 atype 和 count 创建基本类型数组，代码如下：

```
func (self *NEW_ARRAY) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    count := stack.PopInt()
    if count < 0 {
        panic("java.lang.NegativeArraySizeException")
    }

    classLoader := frame.Method().Class().Loader()
    arrClass := getPrimitiveArrayClass(classLoader, self.atype)
    arr := arrClass.NewArray(uint(count))
    stack.PushRef(arr)
}
```

如果 count 小于 0，则抛出 NegativeArraySizeException 异常，否则根据 atype 值使用当前类的类加载器加载数组类，然后创建数组对象并推入操作数栈。getPrimitiveArrayClass() 函数的代码如下：

```
func getPrimitiveArrayClass(loader *heap.ClassLoader, atype uint8) *heap.Class {
    switch atype {
    case AT_BOOLEAN:    return loader.LoadClass("[Z")
    case AT_BYTE:       return loader.LoadClass("[B")
    case AT_CHAR:       return loader.LoadClass("[C")
    case AT_SHORT:      return loader.LoadClass("[S")
    case AT_INT:        return loader.LoadClass("[I")
    case AT_LONG:       return loader.LoadClass("[J")
    case AT_FLOAT:      return loader.LoadClass("[F")
    case AT_DOUBLE:     return loader.LoadClass("[D")
    default: panic("Invalid atype!")
    }
}
```

下面实现 anewarray 指令。

8.3.2 anewarray 指令

anewarray 指令用来创建引用类型数组。在 ch08\instructions\references 目录下创建 anewarray.go 文件，在其中定义 anewarray 指令，代码如下：

```
package references

import "jvmgo/ch08/instructions/base"
import "jvmgo/ch08/rtda"
import "jvmgo/ch08/rtda/heap"

// Create new array of reference
type ANEW_ARRAY struct{ base.Index16Instruction }
```

anewarray 指令也需要两个操作数。第一个操作数是 uint16 索引，来自字节码。通过这个索引可以从当前类的运行时常量池中找到一个类符号引用，解析这个符号引用就可以得到数组元素的类。第二个操作数是数组长度，从操作数栈中弹出。Execute() 方法根据数组元素的类型和数组长度创建引用类型数组，代码如下：

```
func (self *ANEW_ARRAY) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    classRef := cp.GetConstant(self.Index).(*rtc.ClassRef)
    componentClass := classRef.ResolvedClass()

    stack := frame.OperandStack()
    count := stack.PopInt()
    if count < 0 {
        panic("java.lang.NegativeArraySizeException")
    }

    arrClass := componentClass.ArrayClass()
    arr := arrClass.NewArray(uint(count))
    stack.PushRef(arr)
}
```

上面的代码比较容易理解，这里就不详细解释了。Class 结构体的 ArrayClass() 方法返回与类对应的数组类，代码在 class.go 文件中，如下所示：

```
func (self *Class) ArrayClass() *Class {
    arrayClassName := getArrayClassName(self.name)
    return self.loader.LoadClass(arrayClassName)
}
```

先根据类名得到数组类名，然后调用类加载器加载数组类即可。在 ch08\rtda\heap 目录下创建 class_name_helper.go 文件，在其中实现 getArrayClassName() 函数，代码如下：

```
package heap

func getArrayClassName(className string) string {
    return "[" + toDescriptor(className)
}
```

把类名转变成类型描述符，然后在前面加上方括号即可。在 class_name_helper.go 文件中实现 toDescriptor() 函数，代码如下：

```
func toDescriptor(className string) string {
    if className[0] == '[' {
        return className
    }
    if d, ok := primitiveTypes[className]; ok {
        return d
    }
    return "L" + className + ";"
}
```

如果是数组类名，描述符就是类名，直接返回即可。如果是基本类型名，返回对应的类型描述符，否则肯定是普通的类名，前面加上方括号，结尾加上分号即可得到类型描述符。primitiveTypes 变量也在 class_name_helper.go 文件中定义，代码如下：

```
var primitiveTypes = map[string]string{
    "void":     "V",
    "boolean":  "Z",
    "byte":     "B",
    "short":    "S",
    "int":      "I",
    "long":     "J",
    "char":     "C",
    "float":    "F",
    "double":   "D",
}
```

除了 newarray 和 anewarray，还有一个 multianewarray 指令，专门用来创建多维数组。这个指令比较复杂，放到最后实现。下面来看 arraylength 指令。

8.3.3　arraylength 指令

arraylength 指令用于获取数组长度。在 ch08\instructions\references 目录下创建 arraylength.go，在其中定义 arraylength 指令，代码如下：

```
package references
```

```
import "jvmgo/ch08/instructions/base"
import "jvmgo/ch08/rtda"

// Get length of array
type ARRAY_LENGTH struct{ base.NoOperandsInstruction }
```

arraylength 指令只需要一个操作数，即从操作数栈顶弹出的数组引用。Execute() 方法把数组长度推入操作数栈顶，代码如下：

```
func (self *ARRAY_LENGTH) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    arrRef := stack.PopRef()
    if arrRef == nil {
        panic("java.lang.NullPointerException")
    }

    arrLen := arrRef.ArrayLength()
    stack.PushInt(arrLen)
}
```

如果数组引用是 null，则需要抛出 NullPointerException 异常，否则取数组长度，推入操作数栈顶即可。下面实现 <t>aload 和 <t>astore 系列指令。

8.3.4 <t>aload 指令

<t>aload 系列指令按索引取数组元素值，然后推入操作数栈。在 ch08\instructions\loads 目录下创建 xaload.go 文件，在其中定义 8 条指令，代码如下：

```
package loads

import "jvmgo/ch08/instructions/base"
import "jvmgo/ch08/rtda"
import "jvmgo/ch08/rtda/heap"

type AALOAD struct{ base.NoOperandsInstruction }
type BALOAD struct{ base.NoOperandsInstruction }
type CALOAD struct{ base.NoOperandsInstruction }
type DALOAD struct{ base.NoOperandsInstruction }
type FALOAD struct{ base.NoOperandsInstruction }
type IALOAD struct{ base.NoOperandsInstruction }
type LALOAD struct{ base.NoOperandsInstruction }
type SALOAD struct{ base.NoOperandsInstruction }
```

这 8 条指令的实现大同小异，为了节约篇幅，以 aaload 指令为例进行说明。其 Execute()

方法如下：

```
func (self *AALOAD) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    index := stack.PopInt()
    arrRef := stack.PopRef()

    checkNotNil(arrRef)
    refs := arrRef.Refs()
    checkIndex(len(refs), index)
    stack.PushRef(refs[index])
}
```

首先从操作数栈中弹出第一个操作数：数组索引，然后弹出第二个操作数：数组引用。如果数组引用是 null，则抛出 NullPointerException 异常。这个判断在 checkNotNil() 函数中，代码如下：

```
func checkNotNil(ref *heap.Object) {
    if ref == nil {
        panic("java.lang.NullPointerException")
    }
}
```

如果数组索引小于 0，或者大于等于数组长度，则抛出 ArrayIndexOutOfBoundsException。这个检查在 checkIndex() 函数中，代码如下：

```
func checkIndex(arrLen int, index int32) {
    if index < 0 || index >= int32(arrLen) {
        panic("ArrayIndexOutOfBoundsException")
    }
}
```

如果一切正常，则按索引取出数组元素，推入操作数栈顶。

8.3.5　<t>astore 指令

<t>astore 系列指令按索引给数组元素赋值。在 ch08\instructions\stores 目录下创建 xastore.go 文件，在其中定义 8 条指令，代码如下：

```
package stores

import "jvmgo/ch08/instructions/base"
import "jvmgo/ch08/rtda"
import "jvmgo/ch08/rtda/heap"
```

```
type AASTORE struct{ base.NoOperandsInstruction }
type BASTORE struct{ base.NoOperandsInstruction }
type CASTORE struct{ base.NoOperandsInstruction }
type DASTORE struct{ base.NoOperandsInstruction }
type FASTORE struct{ base.NoOperandsInstruction }
type IASTORE struct{ base.NoOperandsInstruction }
type LASTORE struct{ base.NoOperandsInstruction }
type SASTORE struct{ base.NoOperandsInstruction }
```

这 8 条指令的实现是大同小异，以 iastore 为例进行说明，其 Execute() 方法如下：

```
func (self *IASTORE) Execute(frame *rtda.Frame) {
    stack := frame.OperandStack()
    val := stack.PopInt()
    index := stack.PopInt()
    arrRef := stack.PopRef()

    checkNotNil(arrRef)
    ints := arrRef.Ints()
    checkIndex(len(ints), index)
    ints[index] = int32(val)
}
```

iastore 指令的三个操作数分别是：要赋给数组元素的值、数组索引、数组引用，依次从操作数栈中弹出。如果数组引用是 null，则抛出 NullPointerException。如果数组索引小于 0 或者大于等于数组长度，则抛出 ArrayIndexOutOfBoundsException 异常。这两个检查和 <t>aload 系列指令一样。如果一切正常，则按索引给数组元素赋值。

<t>aload 和 <t>astore 指令实现好了，下面来看 multianewarray 指令。

8.3.6 multianewarray 指令

multianewarray 指令创建多维数组。在 ch08\instructions\references 目录下创建 multianewarray.go 文件，在其中定义 multianewarray 指令，代码如下所示：

```
package references

import "jvmgo/ch08/instructions/base"
import "jvmgo/ch08/rtda"
import "jvmgo/ch08/rtda/heap"

// Create new multidimensional array
type MULTI_ANEW_ARRAY struct {
    index       uint16
    dimensions  uint8
```

```
}
```

multianewarray 指令的第一个操作数是个 uint16 索引，通过这个索引可以从运行时常量池中找到一个类符号引用，解析这个引用就可以得到多维数组类。第二个操作数是个 uint8 整数，表示数组维度。这两个操作数在字节码中紧跟在指令操作码后面，由 FetchOperands() 方法读取，代码如下：

```
func (self *MULTI_ANEW_ARRAY) FetchOperands(reader *base.BytecodeReader) {
    self.index = reader.ReadUint16()
    self.dimensions = reader.ReadUint8()
}
```

multianewarray 指令还需要从操作数栈中弹出 *n* 个整数，分别代表每一个维度的数组长度。Execute() 方法根据数组类、数组维度和各个维度的数组长度创建多维数组，代码如下：

```
func (self *MULTI_ANEW_ARRAY) Execute(frame *rtda.Frame) {
    cp := frame.Method().Class().ConstantPool()
    classRef := cp.GetConstant(uint(self.index)).(*heap.ClassRef)
    arrClass := classRef.ResolvedClass()

    stack := frame.OperandStack()
    counts := popAndCheckCounts(stack, int(self.dimensions))
    arr := newMultiDimensionalArray(counts, arrClass)
    stack.PushRef(arr)
}
```

这里提醒读者注意，在 anewarray 指令中，解析类符号引用后得到的是数组元素的类，而这里解析出来的直接就是数组类。popAndCheckCounts() 函数从操作数栈中弹出 *n* 个 int 值，并且确保它们都大于等于 0。如果其中任何一个小于 0，则抛出 NegativeArraySizeException 异常。代码如下：

```
func popAndCheckCounts(stack *rtda.OperandStack, dimensions int) []int32 {
    counts := make([]int32, dimensions)
    for i := dimensions - 1; i >= 0; i-- {
        counts[i] = stack.PopInt()
        if counts[i] < 0 {
            panic("java.lang.NegativeArraySizeException")
        }
    }
    return counts
}
```

newMultiArray() 函数创建多维数组，代码如下：

```
func newMultiDimensionalArray(counts []int32, arrClass *heap.Class) *heap.Object {
    count := uint(counts[0])
    arr := arrClass.NewArray(count)

    if len(counts) > 1 {
        refs := arr.Refs()
        for i := range refs {
            refs[i] = newMultiDimensionalArray (counts[1:], arrClass.ComponentClass())
        }
    }

    return arr
}
```

Class 结构体的 ComponentClass() 方法返回数组类的元素类型在 array_class.go 文件中，代码如下：

```
func (self *Class) ComponentClass() *Class {
    componentClassName := getComponentClassName(self.name)
    return self.loader.LoadClass(componentClassName)
}
```

ComponentClass() 方法先根据数组类名推测出数组元素类名，然后用类加载器加载元素类即可。getComponentClassName() 函数在 ch08\rtda\heap\class_name_helper.go 文件中，代码如下：

```
func getComponentClassName(className string) string {
    if className[0] == '[' {
        componentTypeDescriptor := className[1:]
        return toClassName(componentTypeDescriptor)
    }
    panic("Not array: " + className)
}
```

数组类名以方括号开头，把它去掉就是数组元素的类型描述符，然后把类型描述符转成类名即可。toClassName() 函数也在 class_name_helper.go 文件中，代码如下：

```
func toClassName(descriptor string) string {
    if descriptor[0] == '[' { // array
        return descriptor
    }
    if descriptor[0] == 'L' { // object
        return descriptor[1 : len(descriptor)-1]
    }
    for className, d := range primitiveTypes {
        if d == descriptor { // primitive
```

```
            return className
        }
    }
    panic("Invalid descriptor: " + descriptor)
}
```

如果类型描述符以方括号开头，那么肯定是数组，描述符即是类名。如果类型描述符以 L 开头，那么肯定是类描述符，去掉开头的 L 和末尾的分号即是类名，否则判断是否是基本类型的描述符，如果是，返回基本类型名称，否则调用 panic() 函数终止程序执行。

至此，multianewarray 终于解释完了。由于该指令比较难理解，用一个例子分析。

```
public void test() {
    int[][][] x = new int[3][4][5];
}
```

上面的 Java 方法创建了一个三维数组，编译之后的字节码如下：

```
00  iconst_3
01  iconst_4
02  iconst_5
03  multianewarray #5 // [[[I, 3
07  astore_1
08  return
```

编译器先生成了三条 iconst_n 指令，然后又生成了一条 multianewarray 指令，剩下的两条指令和数组创建无关。multianewarray 指令的第一个操作数是 5，是个类引用，类名是 [[[I，说明要创建的是 int[][][] 数组。第二个操作数是 3，说明要创建三维数组。

当方法执行时，三条 iconst_n 指令先后把整数 3、4、5 推入操作数栈顶。multianewarray 指令在解码时就已经拿到常量池索引（5）和数组维度（3）。在执行时，它先查找运行时常量池索引，知道要创建的是 int[][][] 数组，接着从操作数栈中弹出三个 int 值，依次是 5、4、3。现在 multianewarray 指令拿到了全部信息，从最外维开始创建数组实例即可。

专门用于数组的指令实现好了，但别忘了还需要修改 ch08\instructions\factory.go 文件，在其中添加这些指令的 case 语句。改动比较简单，这里就不给出代码了。下面修改 instanceof 和 checkcast，让这两条指令可以正确用于数组对象。

8.3.7　完善 instanceof 和 checkcast 指令

虽然说是完善 instanceof 和 checkcast 指令，但实际上这两条指令的代码都没有任何变化。需要修改的是 ch08\rtda\heap\class_hierarchy.go 文件中的 isAssignableFrom() 方法，而

且改动很大，代码如下：

```
func (self *Class) isAssignableFrom(other *Class) bool {
    s, t := other, self
    if s == t {
        return true
    }
    if !s.IsArray() {
        if !s.IsInterface() {
            if !t.IsInterface() {
                return s.IsSubClassOf(t)
            } else {
                return s.IsImplements(t)
            }
        } else {
            if !t.IsInterface() {
                return t.isJlObject()
            } else {
                return t.isSuperInterfaceOf(s)
            }
        }
    } else {
        if !t.IsArray() {
            if !t.IsInterface() {
                return t.isJlObject()
            } else {
                return t.isJlCloneable() || t.isJioSerializable()
            }
        } else {
            sc := s.ComponentClass()
            tc := t.ComponentClass()
            return sc == tc || tc.isAssignableFrom(sc)
        }
    }
    return false
}
```

注意，粗体部分是原来的代码，其余都是新增代码。由于篇幅限制，就不详细解释这个函数了，请读者阅读 Java 虚拟机规范的 8.6.5 节对 instanceof 和 checkcast 指令的描述。需要注意的是：

- 数组可以强制转换成 Object 类型（因为数组的超类是 Object）。
- 数组可以强制转换成 Cloneable 和 Serializable 类型（因为数组实现了这两个接口）。
- 如果下面两个条件之一成立，类型为 []SC 的数组可以强制转换成类型为 []TC 的数组：
  - TC 和 SC 是同一个基本类型。

- TC 和 SC 都是引用类型，且 SC 可以强制转换成 TC。

8.4　测试数组

数组相关的内容差不多都准备好了，下面用经典的冒泡排序算法测试。

```
package jvmgo.book.ch08;

public class BubbleSortTest {
    public static void main(String[] args) {
        int[] arr = {22, 84, 77, 11, 95,  9, 78, 56, 36, 97, 65, 36, 10, 24 ,92, 48};
        bubbleSort(arr);
        printArray(arr);
    }
    private static void bubbleSort(int[] arr) {
        boolean swapped = true;
        int j = 0;
        int tmp;
        while (swapped) {
            swapped = false;
            j++;
            for (int i = 0; i < arr.length - j; i++) {
                if (arr[i] > arr[i + 1]) {
                    tmp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = tmp;
                    swapped = true;
                }
            }
        }
    }
    private static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.println(i);
        }
    }
}
```

打开命令行窗口，执行下面的命令编译本章代码：

```
go install jvmgo\ch08
```

命令执行完毕后，在 D:\go\workspace\bin 目录下出现 ch08.exe 文件。用 javac 编译，然后用 ch08.exe 执行 BubbleSortTest 类，结果如图 8-1 所示。

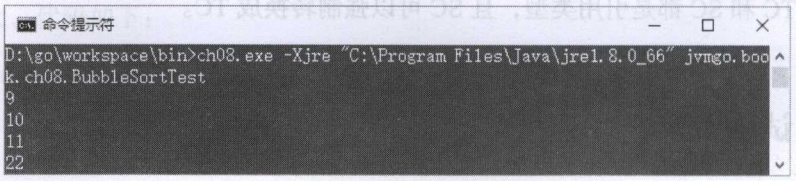


图 8-1 BubbleSortTest 程序执行结果

8.5 字符串

在 class 文件中，字符串是以 MUTF8 格式保存的，这一点在 3.3.7 节讨论过。在 Java 虚拟机运行期间，字符串以 java.lang.String（后面简称 String）对象的形式存在，而在 String 对象内部，字符串又是以 UTF16 格式保存的。字符串相关功能大部分都是由 String（和 StringBuilder 等）类提供的，本节只实现一些辅助功能即可。

String 类有两个实例变量。其中一个是 value，类型是字符数组，用于存放 UTF16 编码后的字符序列。另一个是 hash，缓存计字符串的哈希码，代码如下：

```
package java.lang;

public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
    /** Cache the hash code for the string */
    private int hash; // Default to 0
    ... // 其他代码
}
```

字符串对象是不可变（immutable）的，一旦构造好之后，就无法再改变其状态（这里指 value 字段）。String 类有很多构造函数，其中一个是根据字符数组来创建 String 实例，代码如下：

```
public String(char value[]) {
    this.value = Arrays.copyOf(value, value.length);
}
```

本节将参考上面的构造函数，直接创建 String 实例。为了节约内存，Java 虚拟机内部维护了一个字符串池。String 类提供了 intern() 实例方法，可以把自己放入字符串池。代码如下：

```
public native String intern();
```

本节将实现字符串池，由于 intern() 是本地方法，所以留到第 9 章实现。

8.5.1　字符串池

在 ch08\rtda\heap 目录下创建 string_pool.go 文件，在其中定义 internedStrings 变量，代码如下：

```
package heap

import "unicode/utf16"

var internedStrings = map[string]*Object{}
```

用 map 来表示字符串池，key 是 Go 字符串，value 是 Java 字符串。继续编辑 string_pool.go 文件，在其中实现 JString() 函数，代码如下：

```
func JString(loader *ClassLoader, goStr string) *Object {
    if internedStr, ok := internedStrings[goStr]; ok {
        return internedStr
    }

    chars := stringToUtf16(goStr)
    jChars := &Object{loader.LoadClass("[C"), chars}

    jStr := loader.LoadClass("java/lang/String").NewObject()
    jStr.SetRefVar("value", "[C", jChars)

    internedStrings[goStr] = jStr
    return jStr
}
```

JString() 函数根据 Go 字符串返回相应的 Java 字符串实例。如果 Java 字符串已经在池中，直接返回即可，否则先把 Go 字符串（UTF8 格式）转换成 Java 字符数组（UTF16 格式），然后创建一个 Java 字符串实例，把它的 value 变量设置成刚刚转换而来的字符数组，最后把 Java 字符串放入池中。注意，这里其实是跳过了 String 的构造函数，直接用 hack 的方式创建实例。在前面分析过 String 类的代码，这样做虽然有点投机取巧，但确实是没有问题的。

继续编辑 string_pool.go 文件文件，实现 stringToUtf16() 函数，代码如下：

```
func stringToUtf16(s string) []uint16 {
```

```
    runes := []rune(s) // utf32
    return utf16.Encode(runes)
}
```

Go 语言字符串在内存中是 UTF8 编码的，先把它强制转成 UTF32，然后调用 utf16 包的 Encode() 函数编码成 UTF16。Object 结构体的 SetRefVar() 方法直接给对象的引用类型实例变量赋值，代码如下（在 object.go 文件中）：

```
func (self *Object) SetRefVar(name, descriptor string, ref *Object) {
    field := self.class.getField(name, descriptor, false)
    slots := self.data.(Slots)
    slots.SetRef(field.slotId, ref)
}
```

Class 结构体的 getField() 函数根据字段名和描述符查找字段，代码如下（代码在 class.go 文件中）：

```
func (self *Class) getField(name, descriptor string, isStatic) *Field {
    for c := self; c != nil; c = c.superClass {
        for _, field := range c.fields {
            if field.IsStatic() == isStatic &&
                field.name == name && field.descriptor == descriptor {

                return field
            }
        }
    }
    return nil
}
```

字符串池实现好了，下面修改 ldc 指令和类加载器，让它们支持字符串。

8.5.2 完善 ldc 指令

打开 ch08\instructions\constants\ldc.go 文件，添加一条 import 语句，代码如下：

```
package constants

import "jvmgo/ch08/instructions/base"
import "jvmgo/ch08/rtda"
import "jvmgo/ch08/rtda/heap"
```

然后修改 _ldc() 函数，改动如下：

```
func _ldc(frame *rtda.Frame, index uint) {
```

```
    stack := frame.OperandStack()
    class := frame.Method().Class()
    c := class.ConstantPool().GetConstant(index)

    switch c.(type) {
    case int32: stack.PushInt(c.(int32))
    case float32: stack.PushFloat(c.(float32))
    case string:
        internedStr := heap.JString(class.Loader(), c.(string))
        stack.PushRef(internedStr)
    ... // 其他代码不变
}
```

如果 ldc 试图从运行时常量池中加载字符串常量，则先通过常量拿到 Go 字符串，然后把它转成 Java 字符串实例并把引用推入操作数栈顶。

8.5.3　完善类加载器

打开 ch08\rtda\heap\class_loader.go 文件，修改 initStaticFinalVar 函数，改动如下：

```
func initStaticFinalVar(class *Class, field *Field) {
    vars := class.staticVars
    cp := class.constantPool
    cpIndex := field.ConstValueIndex()
    slotId := field.SlotId()

    if cpIndex > 0 {
        switch field.Descriptor() {
        ... // 其他 case 语句不变
        case "Ljava/lang/String;":
            goStr := cp.GetConstant(cpIndex).(string)
            jStr := JString(class.Loader(), goStr)
            vars.SetRef(slotId, jStr)
        }
    }
}
```

这里增加了字符串类型静态常量的初始化逻辑，代码比较简单，就不多解释了。至此，字符串相关的工作都做完了，下面进行测试。

8.6　测试字符串

打开 ch08\main.go 文件，修改 startJVM() 函数。改动非常小，只是在调用 interpret()

函数时，把传递给 Java 主方法的参数传递给它，代码如下：

```
func startJVM(cmd *Cmd) {
    ... // 其他代码不变
    if mainMethod != nil {
        interpret(mainMethod, cmd.verboseInstFlag, cmd.args)
    } else {
        fmt.Printf("Main method not found in class %s\n", cmd.class)
    }
}
```

打开 interpreter.go 文件，修改 interpret() 函数，改动如下：

```
func interpret(method *heap.Method, logInst bool, args []string) {
    thread := rtda.NewThread()
    frame := thread.NewFrame(method)
    thread.PushFrame(frame)

    jArgs := createArgsArray(method.Class().Loader(), args)
    frame.LocalVars().SetRef(0, jArgs)

    defer catchErr(thread)
    loop(thread, logInst)
}
```

interpret() 函数接收从 startJVM() 函数中传递过来的 args 参数，然后调用 createArgsArray() 函数把它转换成 Java 字符串数组，最后把这个数组推入操作数栈顶。至此，通过命令行传递给 Java 程序的参数终于可以派上用场了！ createArgsArray() 函数的代码如下：

```
func createArgsArray(loader *heap.ClassLoader, args []string) *heap.Object {
    stringClass := loader.LoadClass("java/lang/String")
    argsArr := stringClass.ArrayClass().NewArray(uint(len(args)))
    jArgs := argsArr.Refs()
    for i, arg := range args {
        jArgs[i] = heap.JString(loader, arg)
    }
    return argsArr
}
```

最后，打开 ch08\instructions\references\invokevirtual.go，修改 _println() 函数，让它可以打印字符串，改动如下：

```
// hack!
func _println(stack *rtda.OperandStack, descriptor string) {
    switch descriptor {
    ... // 其他 case 语句不变
    case "(Ljava/lang/String;)V":
        jStr := stack.PopRef()
        goStr := heap.GoString(jStr)
        fmt.Println(goStr)
    ... // 其他代码不变
}
```

GoString() 函数在 string_pool.go 文件中，代码如下：

```
func GoString(jStr *Object) string {
    charArr := jStr.GetRefVar("value", "[C")
    return utf16ToString(charArr.Chars())
}
```

先拿到 String 对象的 value 变量值，然后把字符数组转换成 Go 字符串。Object 结构体的 GetRefVar() 方法（在 object.go 文件中）如下：

```
func (self *Object) GetRefVar(name, descriptor string) *Object {
    field := self.class.getField(name, descriptor, false)
    slots := self.data.(Slots)
    return slots.GetRef(field.slotId)
}
```

utf16ToString() 函数在 string_pool.go 文件中，代码如下：

```
func utf16ToString(s []uint16) string {
    runes := utf16.Decode(s) // utf8
    return string(runes)
}
```

先把 UTF16 数据转换成 UTF8 编码，然后强制转换成 Go 字符串即可。一切就绪！重新编译本章代码，然后用 ch08.exe 执行在第 1 章就已经出现过的 HelloWorld 程序。

```
package jvmgo.book.ch01;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

久违的“Hello, world!”终于出现在控制台上了，如图 8-2 所示。

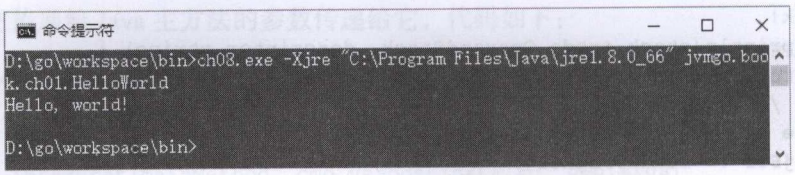


图 8-2 HelloWorld 程序执行结果

再执行一个稍微复杂一些的程序：

```
package jvmgo.book.ch08;

public class PrintArgs {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

执行结果如图 8-3 所示。

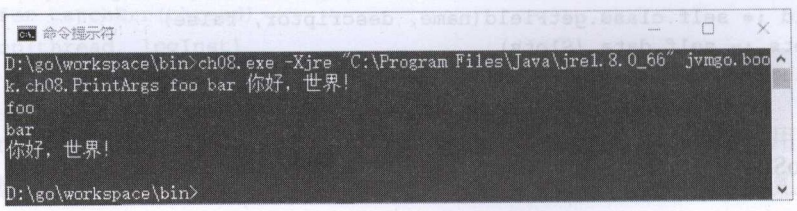
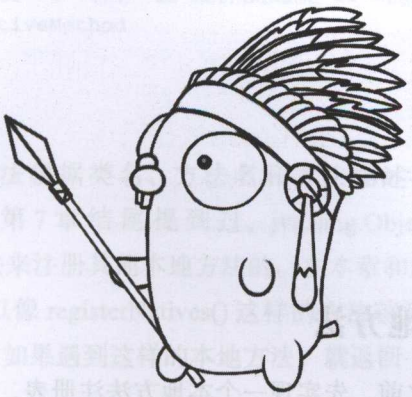


图 8-3 PrintArgs 程序执行结果

8.7 本章小结

本章实现了数组和字符串，在本章的结尾，终于可以运行 HelloWorld 程序了。不过美中不足的是，我们并不是通过调用 System.out.println() 方法，而是通过 hack 的方式打印的。请读者不要着急，下一章会讨论本地方法调用，第 10 章会讨论异常处理。到了第 11 章，将最终去掉这个 hack，让 println() 方法真正得以调用！



第 9 章　本地方法调用

在前面的 8 章里，我们一直在实现 Java 虚拟机的基本功能。我们已经知道，要想运行 Java 程序，除了 Java 虚拟机之外，还需要 Java 类库的配合。Java 虚拟机和 Java 类库一起构成了 Java 运行时环境。Java 类库主要用 Java 语言编写，一些无法用 Java 语言实现的方法则使用本地语言编写，这些方法叫作本地方法。从本章开始，将陆续实现一些 Java 类库中的本地方法。

OpenJDK 类库中的本地方法是用 JNI（Java Native Interface）[⊖]编写的，但是要让虚拟机支持 JNI 规范还需要做大量的工作。由于本书的主要目的是介绍 Java 虚拟机的工作原理，为了不陷入 JNI 规范的细节之中，将使用 Go 语言来实现这些方法。

开始编写代码之前，还是先把目录结构准备好。复制 ch08 目录，改名为 ch09。修改 main.go 等文件，把 import 语句中的 ch08 全都替换成 ch09。在 ch09 目录下创建 native 子目录，本章新增的 go 文件主要都在这个目录（和它的子目录）中。现在，目录结构看起来是下面这个样子：

```
D:\go\workspace\src
  |-jvmgo
    |-ch01 ~ ch08
    |-ch09
```

⊖ http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html

```
|-classfile
|-classpath
|-instructions
|-native
|-rtda
  |-cmd.go
  |-interpreter.go
  |-main.go
```

9.1 注册和查找本地方法

在开始实现本地方法之前，先实现一个本地方法注册表，用来注册和查找本地方法。在 ch09\native 目录下创建 registry.go，先在其中定义 NativeMethod 类型和 registry 变量，代码如下：

```
package native

import "jvmgo/ch09/rtda"

type NativeMethod func(frame *rtda.Frame)

var registry = map[string]NativeMethod{}
```

把本地方法定义成一个函数，参数是 Frame 结构体指针，没有返回值。这个 frame 参数就是本地方法的工作空间，也就是连接 Java 虚拟机和 Java 类库的桥梁，后面会看到它是如何发挥作用的。registry 变量是个哈希表，值是具体的本地方法实现。那么键是什么呢？继续编辑 registry.go 文件，在其中实现 Register() 函数，代码如下：

```
func Register(className, methodName, methodDescriptor string, method NativeMethod) {
    key := className + "~" + methodName + "~" + methodDescriptor
    registry[key] = method
}
```

类名、方法名和方法描述符加在一起才能唯一确定一个方法，所以把它们的组合作为本地方法注册表的键，Register() 函数把前述三种信息和本地方法实现关联起来。继续编辑 registry.go 文件，在其中实现 FindNativeMethod () 方法，代码如下：

```
func FindNativeMethod(className, methodName, methodDescriptor string) NativeMethod {
    key := className + "~" + methodName + "~" + methodDescriptor
    if method, ok := registry[key]; ok {
        return method
    }
```

```
    if methodDescriptor == "()V" && methodName == "registerNatives" {
        return emptyNativeMethod
    }
    return nil
}
```

FindNativeMethod() 方法根据类名、方法名和方法描述符查找本地方法实现，如果找不到，则返回 nil。第 7 章结尾提到过，jva.lang.Object 等类是通过一个叫作 registerNatives() 的本地方法来注册其他本地方法的。在本章和后面的章节中，将自己注册所有的本地方法实现。所以像 registerNatives() 这样的方法就没有太大的用处。为了避免重复代码，这里统一处理，如果遇到这样的本地方法，就返回一个空的实现，代码如下：

```
func emptyNativeMethod(frame *rtda.Frame) {
    // do nothing
}
```

本地方法注册表准备好了，下面介绍如何调用本地方法。

9.2　调用本地方法

第 7 章用一段 hack 代码来跳过本地方法的执行。现在，终于可以把这段代码删除了！编辑 ch09\instructions\base\method_invoke_logic.go，把 fmt 包的导入语句和 InvokeMethod() 函数中的 hack 代码删除。为了节约篇幅，这里就不给出代码了。

Java 虚拟机规范并没有规定如何实现和调用本地方法，这给了我们充分的空间来发挥自己的想象力。读者很快就会看到，我们将利用 Java 虚拟机栈执行本地方法，所以除了删除上面的 InvokeMethod() 函数中的 hack 代码之外，不用做任何修改。

但是，本地方法并没有字节码，如何利用 Java 虚拟机栈来执行呢？ Java 虚拟机规范预留了两条指令，操作码分别是 0xFE 和 0xFF。下面将使用 0xFE 指令来达到这个目的。打开 ch09\rtda\heap\method.go 文件，修改 newMethods () 函数，改动如下：

```
func newMethods(class *Class, cfMethods []*classfile.MemberInfo) []*Method {
    methods := make([]*Method, len(cfMethods))
    for i, cfMethod := range cfMethods {
        methods[i] = newMethod(class, cfMethod)
    }
    return methods
}
```

为了避免 newMethods() 函数变得太长，我们抽取出一个 newMethod() 函数，代码如下：

```
func newMethod(class *Class, cfMethod *classfile.MemberInfo) *Method {
    method := &Method{}
    method.class = class
    method.copyMemberInfo(cfMethod)
    method.copyAttributes(cfMethod)
    md := parseMethodDescriptor(method.descriptor)
    method.calcArgSlotCount(md.parameterTypes)
    if method.IsNative() {
        method.injectCodeAttribute(md.returnType)
    }
    return method
}
```

粗体部分需要解释一下：先计算 argSlotCount 字段，如果是本地方法，则注入字节码和其他信息。继续编辑 method.go 文件，添加 injectCodeAttribute() 方法，代码如下：

```
func (self *Method) injectCodeAttribute(returnType string) {
    self.maxStack = 4
    self.maxLocals = self.argSlotCount
    switch returnType[0] {
    case 'V': self.code = []byte{0xfe, 0xb1} // return
    case 'D': self.code = []byte{0xfe, 0xaf} // dreturn
    case 'F': self.code = []byte{0xfe, 0xae} // freturn
    case 'J': self.code = []byte{0xfe, 0xad} // lreturn
    case 'L', '[': self.code = []byte{0xfe, 0xb0} // areturn
    default: self.code = []byte{0xfe, 0xac} // ireturn
    }
}
```

本地方法在 class 文件中没有 Code 属性，所以需要给 maxStack 和 maxLocals 字段赋值。本地方法帧的操作数栈至少要能容纳返回值，为了简化代码，暂时给 maxStack 字段赋值为 4。因为本地方法帧的局部变量表只用来存放参数值，所以把 argSlotCount 赋给 maxLocals 字段刚好。至于 code 字段，也就是本地方法的字节码，第一条指令都是 0xFE，第二条指令则根据函数的返回值选择相应的返回指令。

另外，由于把方法描述符的解析挪到了 newMethod() 函数中，所以 calcArgSlotCount() 方法也稍微有些变化（增加了一个参数），变动如下：

```
func (self *Method) calcArgSlotCount(paramTypes []string) {
    for _, paramType := range paramTypes {
    ... // 其他代码不变
}
```

下面我们来实现 0xFE 指令。在 ch09\instructions 目录下创建 reserved 子目录，然后在该目录下创建 invokenative.go 文件，在其中定义 0xFE（后面称之为 invokenative）指令，代码如下：

```
package reserved

import "jvmgo/ch09/instructions/base"
import "jvmgo/ch09/rtda"
import "jvmgo/ch09/native"

type INVOKE_NATIVE struct{ base.NoOperandsInstruction }
```

这个指令不需要操作数，Execute() 方法的代码如下：

```
func (self *INVOKE_NATIVE) Execute(frame *rtda.Frame) {
    method := frame.Method()
    className := method.Class().Name()
    methodName := method.Name()
    methodDescriptor := method.Descriptor()

    nativeMethod := native.FindNativeMethod(className, methodName, methodDescriptor)
    if nativeMethod == nil {
        methodInfo := className + "." + methodName + methodDescriptor
        panic("java.lang.UnsatisfiedLinkError: " + methodInfo)
    }

    nativeMethod(frame)
}
```

根据类名、方法名和方法描述符从本地方法注册表中查找本地方法实现，如果找不到，则抛出 UnsatisfiedLinkError 异常，否则直接调用本地方法。最后，还需要修改 instructions\factory.go 文件，在其中添加 invokenative 指令的 case 语句，这里就不给出代码了。

现在，万事俱备，只欠实现本地方法！接下来，我们将实现 Object 和 String 等类的一些本地方法。在后面几章中，还会实现更多的本地方法。

9.3　反射

Java 的反射机制十分强大，本节讨论的内容只是冰山一角。

9.3.1　类和对象之间的关系

在 Java 中，类也表现为普通的对象，它的类是 java.lang.Class。听起来有点像鸡生蛋

还是蛋生鸡的问题：类也是对象，而对象又是类的实例。那么在 Java 虚拟机内部，究竟是先有类还是先有对象呢？下面就来一探究竟。

如前所述，Java 有强大的反射能力。可以在运行期间获取类的各种信息、存取静态和实例变量、调用方法，等等。要想运用这种能力，获取类对象[⊖]是第一步。在 Java 语言中，有两种方式可以获得类对象引用：使用类字面值和调用对象的 getClass() 方法。下面的 Java 代码演示了这两种方式。

```
System.out.println(String.class);
System.out.println("abc".getClass());
```

在第 6 章中，通过 Object 结构体的 class 字段建立了类和对象之间的单向关系。现在把这个关系补充完整，让它成为双向的。打开 ch09\rtda\heap\class.go 文件，修改 Class 结构体，添加 jClass 字段，改动如下：

```
type Class struct {
    ... // 其他字段
    jClass *Object // java.lang.Class 实例
}
```

通过 jClass 字段，每个 Class 结构体实例都与一个类对象关联。另外需要给 jClass 字段定义 Getter 方法，代码比较简单，就不给出了。下面打开 ch09\rtda\heap\object.go 文件，修改 Object 结构体，添加 extra 字段，改动如下：

```
type Object struct {
    class *Class
    data  interface{}
    extra interface{}
}
```

extra 字段用来记录 Object 结构体实例的额外信息。同样给它定义 Getter 和 Setter 方法，这里就不给出代码了。这个字段之所以是 interface{} 类型，是因为它在后面几章还会有其他用途。本章，只用它来记录类对象对应的 Class 结构体指针。

如果读者读到这里感觉有些吃力，请不要怀疑自己的理解能力，一定是笔者表达得不够好。另外，笔者在写这一节时，自己也是犯了很多次迷糊的。为了帮助大家更好地理解类和对象之间的关系，让我们想象这样一个极简化的 Java 虚拟机运行时状态：方法区中只加载了两个类，java.lang.Object 和 java.lang.Class；堆中只通过 new 指令分配了一个对象。

⊖ 在本书中，类对象特指 java.lang.Class 类的实例；对象泛指任何类的实例。

此时 Java 虚拟机的内存状态如图 9-1 所示。

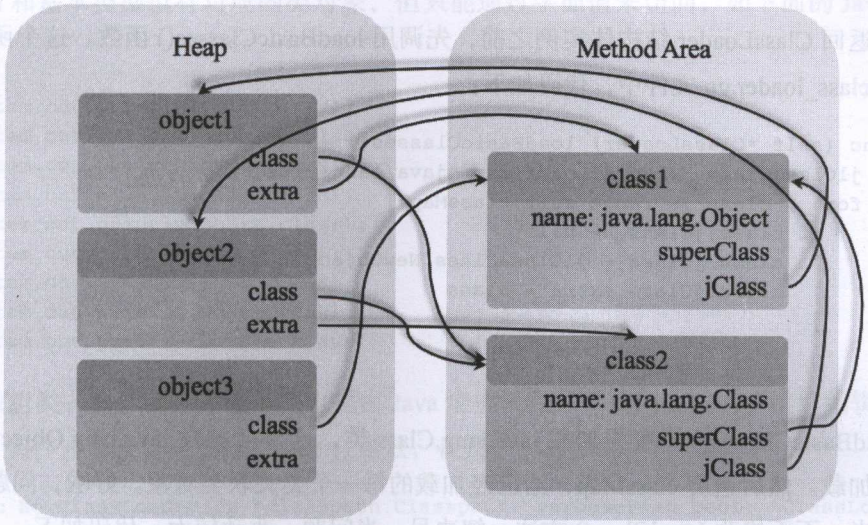


图 9-1　类和对象关系图

图 9-1 只画出了 Class 和 Object 结构体的必要字段，并且刻意分开了堆和方法区。在方法区中，class1 和 class2 分别是 java.lang.Object 和 java.lang.Class 类的数据。在堆中，object1 和 object2 分别是 java.lang.Object 和 java.lang.Class 的类对象。object3 是单独的 java.lang.Object 实例。虽然已经简化到了极点，但仍然有 8 条箭头，希望有密集恐惧症的读者不要被吓倒。

9.3.2　修改类加载器

Class 和 Object 结构体准备好了，接下来修改类加载器，让每一个加载到方法区中的类都有一个类对象与之相关联。打开 ch09\rtda\heap\class_loader.go 文件，修改 NewClassLoader() 函数，改动如下：

```
func NewClassLoader(cp *classpath.Classpath, verboseFlag bool) *ClassLoader {
    loader := &ClassLoader{
        cp:          cp,
        verboseFlag: verboseFlag,
        classMap:    make(map[string]*Class),
    }

    loader.loadBasicClasses()
```

```
    return loader
}
```

在返回 ClassLoader 结构体实例之前，先调用 loadBasicClasses() 函数。这个函数也要添加到 class_loader.go 文件中，代码如下：

```
func (self *ClassLoader) loadBasicClasses() {
    jlClassClass := self.LoadClass("java/lang/Class")
    for _, class := range self.classMap {
        if class.jClass == nil {
            class.jClass = jlClassClass.NewObject()
            class.jClass.extra = class
        }
    }
}
```

loadBasicClasses() 函数先加载 java.lang.Class 类，这又会触发 java.lang.Object 等类和接口的加载。然后遍历 classMap，给已经加载的每一个类关联类对象。好啦，问题已经解决了一半。下面修改 LoadClass() 方法，解决另一半问题。改动较大，代码如下：

```
func (self *ClassLoader) LoadClass(name string) *Class {
    if class, ok := self.classMap[name]; ok {
        return class // already loaded
    }

    var class *Class
    if name[0] == '[' { // array class
        class = self.loadArrayClass(name)
    } else {
        class = self.loadNonArrayClass(name)
    }

    if jlClassClass, ok := self.classMap["java/lang/Class"]; ok {
        class.jClass = jlClassClass.NewObject()
        class.jClass.extra = class
    }

    return class
}
```

主要的变动是粗体部分。在类加载完之后，看 java.lang.Class 是否已经加载。如果是，则给类关联类对象。这样，在 loadBasicClasses() 和 LoadClass() 方法的配合之下，所有加载到方法区的类都设置好了 jClass 字段。

9.3.3　基本类型的类

void 和基本类型也有对应的类对象，但只能通过字面值来访问，如下面的 Java 代码所示。

```
System.out.println(void.class);
System.out.println(boolean.class);
System.out.println(byte.class);
System.out.println(char.class);
System.out.println(short.class);
System.out.println(int.class);
System.out.println(long.class);
System.out.println(float.class);
System.out.println(double.class);
```

和数组类一样，基本类型的类也是由 Java 虚拟机在运行期间生成的。继续编辑 class_loader.go 文件，修改 NewClassLoader() 函数，在其中添加如下一行代码：

```
func NewClassLoader(cp *classpath.Classpath, verboseFlag bool) *ClassLoader {
    ... // 前面的代码不变
    loader.loadBasicClasses()
    loader.loadPrimitiveClasses()
    return loader
}
```

loadPrimitiveClasses() 方法加载 void 和基本类型的类，代码如下：

```
func (self *ClassLoader) loadPrimitiveClasses() {
    for primitiveType, _ := range primitiveTypes {
        self.loadPrimitiveClass(primitiveType) // primitiveType 是 void、int、float 等
    }
}
```

生成 void 和基本类型类的代码在 loadPrimitiveClass() 方法中，代码如下：

```
func (self *ClassLoader) loadPrimitiveClass(className string) {
    class := &Class{
        accessFlags:       ACC_PUBLIC,
        name:              className,
        loader:            self,
        initStarted:       true,
    }
    class.jClass = self.classMap["java/lang/Class"].NewObject()
    class.jClass.extra = class
    self.classMap[className] = class
}
```

这里有三点需要说明。第一，void 和基本类型的类名就是 void、int、float 等。第二，基本类型的类没有超类，也没有实现任何接口。第三，非基本类型的类对象是通过 ldc 指令加载到操作数栈中的，将在 9.3.4 节修改 ldc 指令，让它支持类对象。而基本类型的类对象，虽然在 Java 代码中看起来是通过字面量获取的，但是编译之后的指令并不是 ldc，而是 getstatic。每个基本类型都有一个包装类，包装类中有一个静态常量，叫作 TYPE，其中存放的就是基本类型的类。例如 java.lang.Integer 类，代码如下：

```
public final class Integer extends Number implements Comparable<Integer> {
    ... // 其他代码
    @SuppressWarnings("unchecked")
    public static final Class<Integer>  TYPE
            = (Class<Integer>) Class.getPrimitiveClass("int");
    ... // 其他代码
}
```

也就是说，基本类型的类是通过 getstatic 指令访问相应包装类的 TYPE 字段加载到操作数栈中的。Class.getPrimitiveClass() 是个本地方法，将在 9.3.5 节实现它。包装类将在 9.7 小节详细讨论。

9.3.4 修改 ldc 指令

和基本类型、字符串字面值一样，类对象字面值也是由 ldc 指令加载的。本节修改 ldc 指令，让它可以加载类对象。打开 ch09\instructions\constants\ldc.go 文件，修改 _ldc() 函数，改动如下：

```
func _ldc(frame *rtda.Frame, index uint) {
    stack := frame.OperandStack()
    class := frame.Method().Class()
    c := class.ConstantPool().GetConstant(index)

    switch c.(type) {
    case int32: ...
    case float32: ...
    case string: ...
    case *heap.ClassRef:
        classRef := c.(*heap.ClassRef)
        classObj := classRef.ResolvedClass().JClass()
        stack.PushRef(classObj)
    default: ...
    }
}
```

以上只是增加了一个 case 语句，其他地方没什么变化。如果运行时，常量池中的常量是类引用，则解析类引用，然后把类的类对象推入操作数栈顶。

9.3.5　通过反射获取类名

为了支持通过反射获取类名，本小节将实现以下 4 个本地方法：

- java.lang.Object.getClass()
- java.lang.Class. getPrimitiveClass()
- java.lang.Class. getName0()
- java.lang.Class. desiredAssertionStatus0()

Object.getClass() 就不用多说了，它返回对象的类对象引用。Class.getPrimitiveClass() 在 9.3.3 节提到过，基本类型的包装类在初始化时会调用这个方法给 TPYE 字段赋值。Character 类是基本类型 char 的包装类，它在初始化时会调用 Class.desiredAssertionStatus0() 方法，所以这个方法也需要实现。最后，之所以要实现 getName0() 方法，是因为 Class.getName() 方法是依赖这个本地方法工作的，该方法的代码如下：

```
// java.lang.Class
public String getName() {
    String name = this.name;
    if (name == null)
        this.name = name = getName0();
    return name;
}
```

在 ch09\native 目录下创建 java 子目录，在 java 子目录下创建 lang 子目录，然后在 lang 目录中创建 Object.go 文件，在其中注册 getClass() 本地方法，代码如下：

```
package lang

import "jvmgo/ch09/native"
import "jvmgo/ch09/rtda"

func init() {
    native.Register("java/lang/Object", "getClass", "()Ljava/lang/Class;", getClass)
}
```

继续编辑 Object.go，实现 getClass() 函数，代码如下：

```
// public final native Class<?> getClass();
func getClass(frame *rtda.Frame) {
```

```
    this := frame.LocalVars().GetThis()
    class := this.Class().JClass()
    frame.OperandStack().PushRef(class)
}
```

这是实现的第一个本地方法，所以有必要详细解释一下。首先，从局部变量表中拿到 this 引用。GetThis() 方法其实就是调用 GetRef(0)，不过为了提高代码的可读性，给 LocalVars 结构体添加了这个方法。有了 this 引用后，通过 Class() 方法拿到它的 Class 结构体指针，进而又通过 JClass() 方法拿到它的类对象。最后，把类对象推入操作数栈顶。这样，只用了 3 行代码，Object.getClass() 方法就实现好了。

在 ch09\native\java\lang 目录下创建 Class.go 文件，在其中注册 3 个本地方法，代码如下：

```
package lang

import "jvmgo/ch09/native"
import "jvmgo/ch09/rtda"
import "jvmgo/ch09/rtda/heap"

func init() {
    native.Register("java/lang/Class", "getPrimitiveClass",
        "(Ljava/lang/String;)Ljava/lang/Class;", getPrimitiveClass)
    native.Register("java/lang/Class", "getName0", "()Ljava/lang/String;", getName0)
    native.Register("java/lang/Class", "desiredAssertionStatus0",
        "(Ljava/lang/Class;)Z", desiredAssertionStatus0)
}
```

先实现 getPrimitiveClass() 方法，代码如下：

```
// static native Class<?> getPrimitiveClass(String name);
func getPrimitiveClass(frame *rtda.Frame) {
    nameObj := frame.LocalVars().GetRef(0)
    name := heap.GoString(nameObj)

    loader := frame.Method().Class().Loader()
    class := loader.LoadClass(name).JClass()

    frame.OperandStack().PushRef(class)
}
```

getPrimitiveClass() 是静态方法。先从局部变量表中拿到类名，这是个 Java 字符串，需要把它转成 Go 字符串。基本类型的类已经加载到了方法区中，直接调用类加载器的 LoadClass() 方法获取即可。最后，把类对象引用推入操作数栈顶。下面实现 getName0() 方法，代码如下：

```
// private native String getName0();
func getName0(frame *rtda.Frame) {
    this := frame.LocalVars().GetThis()
    class := this.Extra().(*heap.Class)

    name := class.JavaName()
    nameObj := heap.JString(class.Loader(), name)

    frame.OperandStack().PushRef(nameObj)
}
```

首先从局部变量表中拿到 this 引用，这是一个类对象引用，通过 Extra() 方法可以获得与之对应的 Class 结构体指针。然后拿到类名，转成 Java 字符串并推入操作数栈顶。注意这里需要的是 java.lang.Object 这样的类名，而非 java/lang/Object。Class 结构体的 JavaName() 方法返回转换后的类名，代码如下：

```
func (self *Class) JavaName() string {
    return strings.Replace(self.name, "/", ".", -1)
}
```

本书不讨论断言。desiredAssertionStatus0() 方法把 false 推入操作数栈顶，代码如下：

```
// private static native boolean desiredAssertionStatus0(Class<?> clazz);
func desiredAssertionStatus0(frame *rtda.Frame) {
    frame.OperandStack().PushBoolean(false)
}
```

4 个本地方法都实现好了，而且也已经在 init() 函数中注册，那么可以进行测试了吗？还不行，因为 init() 函数还没有机会执行。编辑 ch09\instructions\reserved\invokenative.go 文件，在其中导入 lang 包，代码如下：

```
package reserved

import "jvmgo/ch09/instructions/base"
import "jvmgo/ch09/rtda"
import "jvmgo/ch09/native"
import _ "jvmgo/ch09/native/java/lang"
```

如果没有任何包依赖 lang 包，它就不会被编译进可执行文件，上面的本地方法也就不会被注册。所以需要一个地方导入 lang 包，把它放在 invokenative.go 文件中。由于没有显示使用 lang 中的变量或函数，所以必须在包名前面加上下划线，否则无法通过编译。这个技术在 Go 语言中叫作“import for side effect”[⊖]。

⊖ https://golang.org/doc/effective_go.html#blank_import

9.3.6 测试本节代码

打开命令行窗口，执行下面的命令编译本章代码：

```
go install jvmgo\ch09
```

命令执行完毕后，在 D:\go\workspace\bin 目录下出现 ch09.exe 文件。用 ch09.exe 运行下面的 Java 程序：

```
package jvmgo.book.ch09;

public class GetClassTest {
    public static void main(String[] args) {
        System.out.println(void.class.getName()); // void
        System.out.println(boolean.class.getName()); // boolean
        System.out.println(byte.class.getName()); // byte
        System.out.println(char.class.getName()); // char
        System.out.println(short.class.getName()); // short
        System.out.println(int.class.getName()); // int
        System.out.println(long.class.getName()); // long
        System.out.println(float.class.getName()); // float
        System.out.println(double.class.getName()); // double
        System.out.println(Object.class.getName()); // java.lang.Object
        System.out.println(int[].class.getName()); // [I
        System.out.println(int[][].class.getName()); // [[I
        System.out.println(Object[].class.getName()); // [Ljava.lang.Object;
        System.out.println(Object[][].class.getName()); // [[Ljava.lang.Object;
        System.out.println(Runnable.class.getName()); // java.lang.Runnable
        System.out.println("abc".getClass().getName()); // java.lang.String
        System.out.println(new double[0].getClass().getName()); // [D
        System.out.println(new String[0].getClass().getName()); //[Ljava.lang.String;
    }
}
```

运行结果如图 9-2 所示。

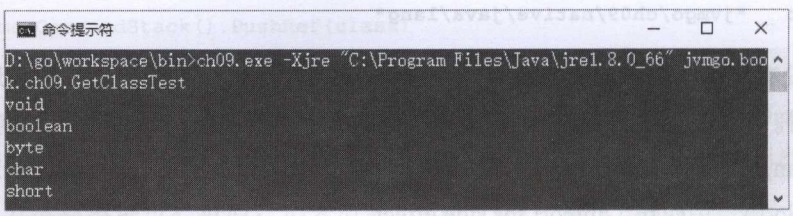


图 9-2 GetClassTest 程序执行结果

9.4　字符串拼接和 String.intern() 方法

9.4.1　Java 类库

在 Java 语言中，通过加号来拼接字符串。作为优化，javac 编辑器会把字符串拼接操作转换成 StringBuilder 的使用。例如下面这段 Java 代码：

```
String hello = "hello,";
String world = "world!";
String str = hello + world;
System.out.println(str);
```

很可能会被 javac 优化为下面这样：

```
String str = new StringBuilder().append("hello,").append("world!").toString();
System.out.println(str);
```

为了运行上面的代码，本节将实现以下 3 个本地方法：

- System.arrayCopy()
- Float. floatToRawIntBits()
- Double. doubleToRawLongBits()

这些方法是在哪里使用的呢？ StringBuilder.append() 方法只是调用了超类的 append() 方法，代码如下：

```
// java.lang.StringBuilder
@Override
public StringBuilder append(String str) {
    super.append(str); // 调用 AbstractStringBuilder.append()
    return this;
}
```

AbstractStringBuilder.append() 方法调用了 String.getChars() 方法获取字符数组，代码如下：

```
// java.lang.AbstractStringBuilder
public AbstractStringBuilder append(String str) {
    if (str == null) return appendNull();
    int len = str.length();
    ensureCapacityInternal(count + len);
    str.getChars(0, len, value, count);
    count += len;
```

```
    return this;
}
```

String.getChars() 方法调用了 System.arraycopy() 方法拷贝数组，代码如下：

```
// java.lang.String
public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin) {
    ... // 其他代码
    System.arraycopy(value, srcBegin, dst, dstBegin, srcEnd - srcBegin);
}
```

StringBuilder.toString() 方法的代码如下：

```
// java.lang.StringBuilder
@Override
public String toString() {
    // Create a copy, don't share the array
     return new String(value, 0, count);
}
```

它调用了 String 的构造函数，这个构造函数调用了 Arrays.copyOfRange() 方法，代码如下：

```
// java.lang.String
public String(char value[], int offset, int count) {
    ... // 其他代码
    this.value = Arrays.copyOfRange(value, offset, offset+count);
}
```

Arrays.copyOfRange() 调用了 Math.min() 方法，代码如下：

```
// java.util.Arrays
public static char[] copyOfRange(char[] original, int from, int to) {
    int newLength = to - from;
    if (newLength < 0) throw new IllegalArgumentException(from + " > " + to);
    char[] copy = new char[newLength];
    System.arraycopy(original, from, copy, 0,
                     Math.min(original.length - from, newLength));
    return copy;
}
```

Math 类在初始化时需要调用 Float.floatToRawIntBits() 和 Double.doubleToRawLongBits() 方法，代码如下：

```
package java.lang;
public final class Math {
    // Use raw bit-wise conversions on guaranteed non-NaN arguments.
```

```
    private static long negativeZeroFloatBits  = Float.floatToRawIntBits(-0.0f);
    private static long negativeZeroDoubleBits = Double.doubleToRawLongBits(-0.0d);
}
```

Java类库介绍完了，下面实现本地方法。

9.4.2 System.arraycopy()方法

在ch09\native\java\lang目录下创建System.go文件，在其中注册arraycopy()方法，代码如下：

```
package lang

import "jvmgo/ch09/native"
import "jvmgo/ch09/rtda"
import "jvmgo/ch09/rtda/heap"

func init() {
    native.Register("java/lang/System", "arraycopy",
        "(Ljava/lang/Object;ILjava/lang/Object;II)V", arraycopy)
}
```

继续编辑System.go，实现arraycopy()方法。代码稍微有些复杂，先来看第一部分。

```
// public static native void arraycopy(
// Object src, int srcPos, Object dest, int destPos, int length)
func arraycopy(frame *rtda.Frame) {
    vars := frame.LocalVars()
    src := vars.GetRef(0)
    srcPos := vars.GetInt(1)
    dest := vars.GetRef(2)
    destPos := vars.GetInt(3)
    length := vars.GetInt(4)
```

先从局部变量表中拿到5个参数。源数组和目标数组都不能是null，否则按照System类的Javadoc应该抛出NullPointerException异常，代码如下：

```
if src == nil || dest == nil {
    panic("java.lang.NullPointerException")
}
```

源数组和目标数组必须兼容才能拷贝，否则应该抛出ArrayStoreExceptio异常，代码如下：

```
if !checkArrayCopy(src, dest) {
```

```
        panic("java.lang.ArrayStoreException")
    }
```

checkArrayCopy() 函数的代码稍后给出。接下来检查 srcPos、destPos 和 length 参数，如果有问题则抛出 IndexOutOfBoundsException 异常，代码如下：

```
    if srcPos < 0 || destPos < 0 || length < 0 ||
        srcPos+length > src.ArrayLength() ||
        destPos+length > dest.ArrayLength() {
        panic("java.lang.IndexOutOfBoundsException")
    }
```

最后，参数合法，调用 ArrayCopy() 函数进行数组拷贝，代码如下：

```
        heap.ArrayCopy(src, dest, srcPos, destPos, length)
    }
```

checkArrayCopy() 函数首先确保 src 和 dest 都是数组，然后检查数组类型。如果两者都是引用数组，则可以拷贝，否则两者必须是相同类型的基本类型数组，代码如下：

```
func checkArrayCopy(src, dest *heap.Object) bool {
    srcClass := src.Class()
    destClass := dest.Class()
    if !srcClass.IsArray() || !destClass.IsArray() {
        return false
    }
    if srcClass.ComponentClass().IsPrimitive() ||
        destClass.ComponentClass().IsPrimitive() {
        return srcClass == destClass
    }
    return true
}
```

Class 结构体的 IsPrimitive() 函数判断类是否是基本类型的类，代码如下：

```
func (self *Class) IsPrimitive() bool {
    _, ok := primitiveTypes[self.name]
    return ok
}
```

真正的数组拷贝逻辑在 ch09\rtda\heap\array_object.go 文件中，代码如下：

```
func ArrayCopy(src, dst *Object, srcPos, dstPos, length int32) {
    switch src.data.(type) {
    case ...
    case []int32:
        _src := src.data.([]int32)[srcPos : srcPos+length]
```

```
        _dst := dst.data.([]int32)[dstPos : dstPos+length]
        copy(_dst, _src)
    case []*Object:
        _src := src.data.([]*Object)[srcPos : srcPos+length]
        _dst := dst.data.([]*Object)[dstPos : dstPos+length]
        copy(_dst, _src)
    }
}
```

利用 Go 的内置函数 copy() 进行 slice 拷贝。为了节约篇幅，上面的代码只给出了 int 数组和对象数组的 case 语句，其他情况代码大同小异。

9.4.3　Float. floatToRawIntBits() 和 Double.doubleToRawLongBits() 方法

Float.floatToRawIntBits() 和 Double. doubleToRawLongBits() 返回浮点数的编码，这两个方法大同小异，以 Float 为例进行介绍。在 ch09\native\java\lang 目录下创建 Float.go 文件，在其中注册 floatToRawIntBits() 本地方法，代码如下：

```
package lang

import "math"
import "jvmgo/ch09/native"
import "jvmgo/ch09/rtda"

func init() {
    native.Register("java/lang/Float",
        "floatToRawIntBits", "(F)I", floatToRawIntBits)
}
```

Go 语言的 math 包提供了一个类似函数：Float32bits()，正好可以用来实现 floatToRawIntBits() 方法，代码如下：

```
// public static native int floatToRawIntBits(float value);
func floatToRawIntBits(frame *rtda.Frame) {
    value := frame.LocalVars().GetFloat(0)
    bits := math.Float32bits(value)
    frame.OperandStack().PushInt(int32(bits))
}
```

9.4.4　String.intern() 方法

第 8 章讨论字符串时，实现了字符串池，但它只能在虚拟机内部使用。下面实现 String 类的 intern() 方法，让 Java 类库也可以使用它。在 ch09\native\java\lang 目录下创建

String.go，在其中注册 intern() 方法，代码如下：

```
package lang

import "jvmgo/ch09/native"
import "jvmgo/ch09/rtda"
import "jvmgo/ch09/rtda/heap"

func init() {
    native.Register("java/lang/String", "intern", "()Ljava/lang/String;", intern)
}
```

继续编辑 String.go 文件，实现 intern() 方法，代码如下：

```
// public native String intern();
func intern(frame *rtda.Frame) {
    this := frame.LocalVars().GetThis()
    interned := heap.InternString(this)
    frame.OperandStack().PushRef(interned)
}
```

如果字符串还没有入池，把它放入并返回该字符串，否则找到已入池字符串并返回。这个逻辑在 InternString() 函数中（ch09\rtda\heap\string_pool.go），代码如下：

```
func InternString(jStr *Object) *Object {
    goStr := GoString(jStr)
    if internedStr, ok := internedStrings[goStr]; ok {
        return internedStr
    }
    internedStrings[goStr] = jStr
    return jStr
}
```

字符串相关的本地方法都实现好了，下面我们进行测试。

9.4.5 测试本节代码

下面的 Java 程序对字符串拼接和入池进行了测试。

```
package jvmgo.book.ch09;

public class StringTest {
    public static void main(String[] args) {
        String s1 = "abc1";
        String s2 = "abc1";
        System.out.println(s1 == s2); // true
```

```
        int x = 1;
        String s3 = "abc" + x;
        System.out.println(s1 == s3); // false

        s3 = s3.intern();
        System.out.println(s1 == s3); // true
    }
}
```

重新编译本章代码，然后测试 StringTest 程序，结果如图 9-3 所示。

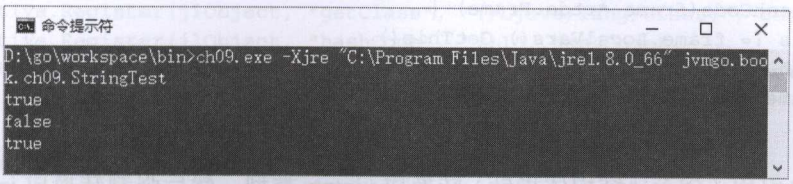


图 9-3　StringTest 程序执行结果

9.5　Object.hashCode()、equals() 和 toString()

Object 类有 3 个非常重要的方法：hashCode() 返回对象的哈希码；equals() 用来比较两个对象是否“相同”；toString() 返回对象的字符串表示。hashCode() 是个本地方法，equals() 和 toString() 则是用 Java 写的，它们的代码如下：

```
package java.lang;
public class Object {
    ... // 其他代码省略
    public native int hashCode();
    public boolean equals(Object obj) {
        return (this == obj);
    }
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
}
```

下面实现 hashCode() 方法。打开 ch09\native\java\lang\Object.go，导入 unsafe 包并注册 hashCode() 方法，代码如下：

```
package lang
```

```
import "unsafe"
import "jvmgo/ch09/native"
import "jvmgo/ch09/rtda"

func init() {
    native.Register("java/lang/Object", "getClass", "()Ljava/lang/Class;", getClass)
    native.Register("java/lang/Object", "hashCode", "()I", hashCode)
}
```

继续编辑 Object.go，实现 hashCode() 方法，代码如下：

```
// public native int hashCode();
func hashCode(frame *rtda.Frame) {
    this := frame.LocalVars().GetThis()
    hash := int32(uintptr(unsafe.Pointer(this)))
    frame.OperandStack().PushInt(hash)
}
```

把对象引用（Object 结构体指针）转换成 uintptr 类型，然后强制转换成 int32 推入操作数栈顶。

本节只实现这一个本地方法。重新编译本章代码，然后测试下面的 Java 程序：

```
package jvmgo.book.ch09;

public class ObjectTest {
    public static void main(String[] args) {
        Object obj1 = new ObjectTest();
        Object obj2 = new ObjectTest();
        System.out.println(obj1.hashCode());
        System.out.println(obj1.toString());
        System.out.println(obj1.equals(obj2));
        System.out.println(obj1.equals(obj1));
    }
}
```

ObjectTest 程序执行结果如图 9-4 所示。

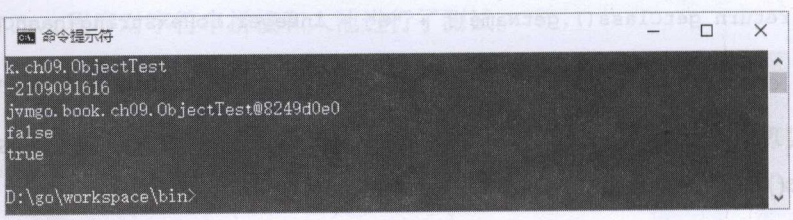


图 9-4　ObjectTest 程序执行结果

9.6　Object.clone()

Object 类提供了 clone() 方法，用来支持对象克隆。这也是一个本地方法，代码如下：

```
// java.lang.Object
protected native Object clone() throws CloneNotSupportedException;
```

本节实现这个方法。在 ch09\native\java\lang\Object.go 文件中注册 clone() 方法，代码如下：

```
func init() {
    native.Register(jlObject, "getClass", "()Ljava/lang/Class;", getClass)
    native.Register(jlObject, "hashCode", "()I", hashCode)
    native.Register(jlObject, "clone", "()Ljava/lang/Object;", clone)
}
```

继续编辑 Object.go，实现 clone() 方法，代码如下：

```
func clone(frame *rtda.Frame) {
    this := frame.LocalVars().GetThis()

    cloneable := this.Class().Loader().LoadClass("java/lang/Cloneable")
    if !this.Class().IsImplements(cloneable) {
        panic("java.lang.CloneNotSupportedException")
    }

    frame.OperandStack().PushRef(this.Clone())
}
```

如果类没有实现 Cloneable 接口，则抛出 CloneNotSupportedException 异常，否则调用 Object 结构体的 Clone() 方法克隆对象，然后把对象副本引用推入操作数栈顶。Clone() 实现稍微有些长，把它放在 ch09\rtda\heap\object_clone.go 文件中，代码如下：

```
func (self *Object) Clone() *Object {
    return &Object{
        class: self.class,
        data:  self.cloneData(),
    }
}
```

数据克隆逻辑在 cloneData() 函数中，代码如下：

```
func (self *Object) cloneData() interface{} {
    switch self.data.(type) {
    case []int8: ...
    case []int16: ...
```

```
    case []uint16: ...
    case []int32: ...
    case []int64: ...
    case []float32: ...
    case []float64: ...
    case []*Object:
        elements := self.data.([]*Object)
        elements2 := make([]*Object, len(elements))
        copy(elements2, elements)
        return elements2
    default: // []Slot
        slots := self.data.(Slots)
        slots2 := newSlots(uint(len(slots)))
        copy(slots2, slots)
        return slots2
    }
}
```

注意，数组也实现了 Cloneable 接口，所以上面代码中的 case 语句针对各种数组进行处理。因为代码都大同小异，所以只给出了引用数组的 case 语句。default 语句对普通对象进行克隆。

重新编译本章代码，测试下面的 Java 程序。

```
package jvmgo.book.ch09;

public class CloneTest implements Cloneable {
    private double pi = 3.14;

    @Override
    public CloneTest clone() {
        try {
            return (CloneTest) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
    }
    public static void main(String[] args) {
        CloneTest obj1 = new CloneTest();
        CloneTest obj2 = obj1.clone();
        obj1.pi = 3.1415926;
        System.out.println(obj1.pi);
        System.out.println(obj2.pi);
    }
}
```

CloneTest 程序执行结果如图 9-5 所示。

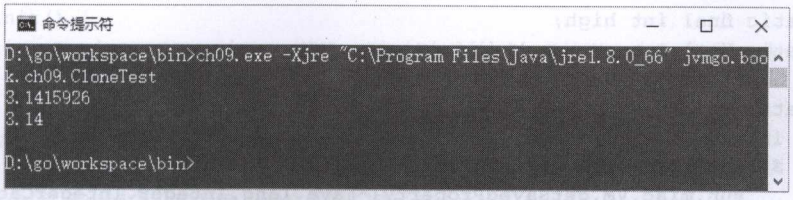


图 9-5　CloneTest 程序执行结果

9.7　自动装箱和拆箱

前面讨论过，为了更好地融入 Java 的对象系统，每种基本类型都有一个包装类与之对应。从 Java 5 开始，Java 语法增加了自动装箱和拆箱（autoboxing/unboxing）能力，可以在必要时把基本类型转换成包装类型或者反之。这个增强完全是由编译器完成的，Java 虚拟机没有做任何调整。

以 int 类型为例，它的包装类是 java.lang.Integer。它提供了 2 个方法来帮助编译器在 int 变量和 Integer 对象之间转换：静态方法 value() 把 int 变量包装成 Integer 对象；实例方法 intValue() 返回被包装的 int 变量。这两个方法的代码如下：

```
package java.lang;
public final class Integer extends Number implements Comparable<Integer> {
    ... // 其他代码省略
    private final int value;
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
    public int intValue() {
        return value;
    }
}
```

由上面的代码可知，Integer.valueOf() 方法并不是每次都创建 Integer() 对象，而是维护了一个缓存池 IntegerCache。对于比较小（默认是 –128 ~ 127）的 int 变量，在 IntegerCache 初始化时就预先加载到了池中，需要用时直接从池里取即可。IntegerCache 是 Integer 类的内部类，为了便于参考，下面给出它的完整代码。

```
private static class IntegerCache {
    static final int low = -128;
```

```
    static final int high;
    static final Integer cache[];

    static {
        int h = 127; // high value may be configured by property
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

具体细节就不解释了，需要说明的是 IntegerCache 在初始化时需要确定缓存池中 Integer 对象的上限值，为此它调用了 sun.misc.VM 类的 getSavedProperty() 方法。要想让 VM 正确初始化需要做很多工作，这个工作推迟到第 11 章进行。这里先用一个 hack 让 VM.getSavedProperty() 方法返回非 null 值，以便 IntegerCache 可以正常初始化。

在 ch09\native 目录下创建 sun\misc 子目录，在其中创建 VM.go 文件，然后在 VM.go 文件中注册 initialize() 方法，代码如下：

```
package misc

import "jvmgo/ch09/instructions/base"
import "jvmgo/ch09/native"
import "jvmgo/ch09/rtda"
import "jvmgo/ch09/rtda/heap"
```

```
func init() {
    native.Register("sun/misc/VM", "initialize", "()V", initialize)
}
```

initialize() 方法的实现如下：

```
// private static native void initialize();
func initialize(frame *rtda.Frame) {
    vmClass := frame.Method().Class()
    savedProps := vmClass.GetRefVar("savedProps", "Ljava/util/Properties;")
    key := heap.JString(vmClass.Loader(), "foo")
    val := heap.JString(vmClass.Loader(), "bar")

    frame.OperandStack().PushRef(savedProps)
    frame.OperandStack().PushRef(key)
    frame.OperandStack().PushRef(val)

    propsClass := vmClass.Loader().LoadClass("java/util/Properties")
    setPropMethod := propsClass.GetInstanceMethod("setProperty",
        "(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/Object;")
    base.InvokeMethod(frame, setPropMethod)
}
```

上面的 hack 代码可能有些不好理解，但翻译成等价的 Java 代码后只有一句话：

```
private static native void initialize() {
    VM.savedProps.setProperty("foo", "bar")
}
```

最后，需要修改 invokenative.go，在其中导入 misc 包，改动如下：

```
package reserved

import "jvmgo/ch09/instructions/base"
import "jvmgo/ch09/rtda"
import "jvmgo/ch09/native"
import _ "jvmgo/ch09/native/java/lang"
import _ "jvmgo/ch09/native/sun/misc"
```

重新编译本章代码，然后测试下面的 Java 程序：

```
package jvmgo.book.ch09;

import java.util.ArrayList;
import java.util.List;

public class BoxTest {
    public static void main(String[] args) {
```

```
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println(list.toString());
        for (int x : list) {
            System.out.println(x);
        }
    }
}
```

BoxTest 程序的执行结果如图 9-6 所示。

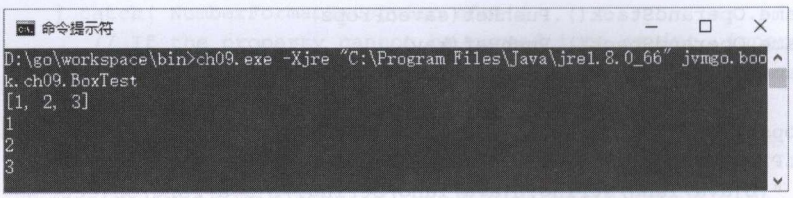


图 9-6 BoxTest 程序执行结果

9.8 本章小结

本章主要讨论了本地方法调用，以及 Java 类库中一些最基本的类。前几章基本上都是围绕 Java 虚拟机本身如何工作而展开讨论。通过本章的学习，读者应该对 Java 虚拟机和 Java 类库如何配合工作有了初步的了解。下一章将讨论异常处理。

第10章 异常处理

异常处理是Java语言非常重要的一个语法，本章从Java虚拟机的角度来讨论异常是如何被抛出和处理的。开始本章之前，还是先把目录结构准备好。复制ch09目录，改名为ch10。修改main.go等文件，把import语句中的ch09全都替换成ch10。本章对目录结构没有太大的调整。

10.1 异常处理概述

在Java语言中，异常可以分为两类：Checked异常和Unchecked异常。Unchecked异常包括java.lang. RuntimeException、java.lang.Error以及它们的子类，其他异常都是Checked异常。所有异常都最终继承自java.lang.Throwable。如果一个方法有可能导致Checked异常抛出，则该方法要么需要捕获该异常并妥善处理，要么必须把该异常列在自己的throws子句中，否则无法通过编译。Unchecked异常没有这个限制。请注意，Java虚拟机规范并没有这个规定，这只是Java语言的语法规则。

异常可以由Java虚拟机抛出，也可以由Java代码抛出。当Java虚拟机在运行过程中遇到比较严重的问题时，会抛出java.lang.Error的某个子类，如StackOverflowError、OutOfMemoryError等。程序一般无法从这种异常里恢复，所以在代码中通常也不必关心

这类异常。一部分指令在执行过程中会导致 Java 虚拟机抛出 java.lang.RuntimeException 的某个子类，如 NullPointerException、IndexOutOfBoundsException 等。这类异常一般是代码中的 bug 导致的，需要格外注意。在代码中抛出和处理异常是由 athrow 指令和方法的异常处理表配合完成的，本章将重点讨论这一点。

在 Java 6 之前，Oracle 的 Java 编译器使用 jsr、jsr_w 和 ret 指令来实现 finally 子句。从 Java 6 开始，已经不再使用这些指令，本章不讨论这三条指令。

10.2 异常抛出

在 Java 代码中，异常是通过 throw 关键字抛出的。Java 虚拟机规范的 3.12 节给了一个例子，代码如下：

```
void cantBeZero(int i) {
   if (i == 0) {
      throw new TestExc();
   }
}
```

上面的方法编译之后，产生下面的字节码：

```
 0  iload_1                    // 把参数 1（i）推入操作数栈顶
 1  ifne 12                    // 如果 i 不等于 0，直接执行 return 指令
 4  new #2                     // 创建 TestExc 实例，把引用推入操作数栈顶
 7  dup                        // 复制 TestExc 实例引用
 8  invokespecial #3           // 调用 TestExc 构造函数（栈顶引用已经作为参数弹出）
11  athrow                     // 抛出异常
12  return                     // 方法返回
```

唯一陌生的指令是 athrow，将在 10.4 节实现该指令。从字节码来看，异常对象似乎也只是普通的对象，通过 new 指令创建，然后调用构造函数进行初始化。这是真的吗？如果查看 java.lang.Exception 或 RuntimeException 的源代码就可以知道，这并不是真的。它们的构造函数都调用了超类 java.lang.Throwable 的构造函数。Throwable 的构造函数又调用了 fillInStackTrace() 方法记录 Java 虚拟机栈信息，这个方法的代码如下：

```
// java.lang.Throwable
public synchronized Throwable fillInStackTrace() {
   if (stackTrace != null ||
       backtrace != null /* Out of protocol state */ ) {
       fillInStackTrace(0);
       stackTrace = UNASSIGNED_STACK;
```

```
    }
    return this;
}
```

fillInStackTrace() 是用 Java 写的，必须借助另外一个本地方法才能访问 Java 虚拟机栈，这个方法就是重载后的 fillInStackTrace(int) 方法，代码如下：

```
private native Throwable fillInStackTrace(int dummy);
```

也就是说，要想抛出异常，Java 虚拟机必须实现这个本地方法。在 10.5 节中，我们会真正实现这个方法，这里先给它一个空的实现。在 ch10\native\java\lang 目录下创建 Throwable.go 文件，在其中注册 fillInStackTrace(int) 方法，代码如下：

```
package lang

import "jvmgo/ch10/native"
import "jvmgo/ch10/rtda"
import "jvmgo/ch10/rtda/heap"
func init() {
    native.Register("java/lang/Throwable", "fillInStackTrace",
        "(I)Ljava/lang/Throwable;", fillInStackTrace)
}

// private native Throwable fillInStackTrace(int dummy);
func fillInStackTrace(frame *rtda.Frame) {
    // 在 10.5 节实现
}
```

异常抛出暂时先讨论到这里，下面介绍如何处理异常。

10.3 异常处理表

异常处理是通过 try-catch 句实现的，还是参考 Java 虚拟机规范的 3.12 节，里面有一个例子，代码如下：

```
void catchOne() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
    }
}
```

上面的方法编译之后，产生下面的字节码：

```
 1  aload_0                 // 把局部变量 0（this）推入操作数栈顶
 2  invokevirtual #4        // 调用 tryItOut() 方法
 4  goto 13                 // 如果 try{} 没有抛出异常，直接执行 return 指令
 7  astore_1                // 否则，异常对象引用在操作数栈顶，把它弹出，并放入局部变量 1
 8  aload_0                 // 把 this 推入栈顶（将作为 handleExc() 方法的参数 0）
 9  aload_1                 // 把异常对象引用推入栈顶（将作为 handleExc() 方法的参数 1）
10  invokevirtual #5        // 调用 handleExc() 方法
13  return                  // 方法返回
```

从字节码来看，如果没有异常抛出，则会直接 goto 到 return 指令，方法正常返回。那么如果有异常抛出，goto 和 return 之间的指令是如何执行的呢？答案是查找方法的异常处理表。异常处理表是 Code 属性的一部分，它记录了方法是否有能力处理某种异常。回顾一下方法的 Code 属性，它的结构如下：

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {   u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

异常处理表的每一项都包含 3 个信息：处理哪部分代码抛出的异常、哪类异常，以及异常处理代码在哪里。具体来说，start_pc 和 end_pc 可以锁定一部分字节码，这部分字节码对应某个可能抛出异常的 try{} 代码块。catch_type 是个索引，通过它可以从运行时常量池中查到一个类符号引用，解析后的类是个异常类，假定这个类是 X。如果位于 start_pc 和 end_pc 之间的指令抛出异常 x，且 x 是 X（或者 X 的子类）的实例，handler_pc 就指出负责异常处理的 catch{} 块在哪里。

回到 catchOne() 方法，它的异常处理表只有如下一项：

start_pc	end_pc	handler_pc	catch_type
0	4	7	TestExc

当tryItOut()方法通过athrow指令抛出TestExc异常时，Java虚拟机首先会查找tryItOut()方法的异常处理表，看它能否处理该异常。如果能，则跳转到相应的字节码开始异常处理。假设tryItOut()方法无法处理异常，Java虚拟机会进一步查看它的调用者，也就是catchOne()方法的异常处理表。catchOne()方法刚好可以处理TestExc异常，使catch{}块得以执行。

假设catchOne()方法也无法处理TestExc异常，Java虚拟机会继续查找catchOne()的调用者的异常处理表。这个过程会一直继续下去，直到找到某个异常处理项，或者到达Java虚拟机栈的底部。把这部分逻辑放在athrow指令中，具体请看10.4小节。下面修改Method结构体，在里面增加异常处理表。

打开ch10\rtda\heap\method.go文件，给Method结构体添加exceptionTable字段，代码如下：

```
type Method struct {
    ... // 其他字段
    exceptionTable ExceptionTable
}
```

然后修改copyAttributes()方法，从Code属性中复制异常处理表，代码如下：

```
func (self *Method) copyAttributes(cfMethod *classfile.MemberInfo) {
    if codeAttr := cfMethod.CodeAttribute(); codeAttr != nil {
        ... // 其他字段
        self.exceptionTable = newExceptionTable(codeAttr.ExceptionTable(),
            self.class.constantPool)
    }
}
```

稍后再介绍ExceptionTable类型和newExceptionTable()函数。继续编辑method.go文件，给Method结构体添加FindExceptionHandler()方法，代码如下：

```
func (self *Method) FindExceptionHandler(exClass *Class, pc int) int {
    handler := self.exceptionTable.findExceptionHandler(exClass, pc)
    if handler != nil {
        return handler.handlerPc
    }
    return -1
}
```

FindExceptionHandler()方法调用ExceptionTable.findExceptionHandler()方法搜索异常处理表，如果能够找到对应的异常处理项，则返回它的handlerPc字段，否则返回-1。

Method 结构体修改完毕，下面来看 ExceptionTable。

在 ch10\rtda\heap 目录下创建 exception_table.go 文件，在其中定义 ExceptionTable 类型，代码如下：

```
package heap

import "jvmgo/ch10/classfile"

type ExceptionTable []*ExceptionHandler
```

ExceptionTable 只是 []*ExceptionHandler 的别名而已，ExceptionHandler 的定义如下：

```
type ExceptionHandler struct {
    startPc     int
    endPc       int
    handlerPc   int
    catchType   *ClassRef
}
```

这 4 个字段在前面已经介绍过，这里不多解释。继续编辑 exception_table.go 文件，在其中实现 newExceptionTable() 函数，代码如下：

```
func newExceptionTable(entries []*classfile.ExceptionTableEntry,
        cp *ConstantPool) ExceptionTable {
    table := make([]*ExceptionHandler, len(entries))
    for i, entry := range entries {
        table[i] = &ExceptionHandler{
            startPc:     int(entry.StartPc()),
            endPc:       int(entry.EndPc()),
            handlerPc:   int(entry.HandlerPc()),
            catchType:   getCatchType(uint(entry.CatchType()), cp),
        }
    }
    return table
}
```

newExceptionTable() 函数把 class 文件中的异常处理表转换成 ExceptionTable 类型。有一点需要特别说明：异常处理项的 catchType 有可能是 0。我们知道 0 是无效的常量池索引，但是在这里 0 并非表示 catch-none，而是表示 catch-all，它的用法马上就会看到。getCatchType() 函数从运行时常量池中查找类符号引用，代码如下：

```
func getCatchType(index uint, cp *ConstantPool) *ClassRef {
    if index == 0 {
        return nil
```

```
    }
    return cp.GetConstant(index).(*ClassRef)
}
```

继续编辑 exception_table.go 文件，实现 findExceptionHandler() 方法，代码如下：

```
func (self ExceptionTable) findExceptionHandler(exClass *Class,
        pc int) *ExceptionHandler {
    for _, handler := range self {
        if pc >= handler.startPc && pc < handler.endPc {
            if handler.catchType == nil {
                return handler // catch-all
            }
            catchClass := handler.catchType.ResolvedClass()
            if catchClass == exClass || catchClass.IsSuperClassOf(exClass) {
                return handler
            }
        }
    }
    return nil
}
```

异常处理表查找逻辑前面已经描述过，此处不再赘述。这里注意两点。第一，startPc 给出的是 try{} 语句块的第一条指令，endPc 给出的则是 try{} 语句块的下一条指令。第二，如果 catchType 是 nil（在 class 文件中是 0），表示可以处理所有异常，这是用来实现 finally 子句的。

为了节约篇幅，本章就不再讨论多个 catch 块、嵌套 try-catch，以及 finally 子句等对应的字节码实现了，读者可以阅读 Java 虚拟机规范的 3.12 节。下一节将实现 athrow 指令。

10.4 实现 athrow 指令

athrow 属于引用类指令，在 ch10\instructions\references 目录下创建 athrow.go 文件，在其中定义 athrow 指令，代码如下：

```
package references

import "reflect"
import "jvmgo/ch10/instructions/base"
import "jvmgo/ch10/rtda"
import "jvmgo/ch10/rtda/heap"
```

```
// Throw exception or error
type ATHROW struct{ base.NoOperandsInstruction }
```

athrow 指令的操作数是一个异常对象引用，从操作数栈弹出。Execute() 方法的代码如下：

```
func (self *ATHROW) Execute(frame *rtda.Frame) {
    ex := frame.OperandStack().PopRef()
    if ex == nil {
        panic("java.lang.NullPointerException")
    }

    thread := frame.Thread()
    if !findAndGotoExceptionHandler(thread, ex) {
        handleUncaughtException(thread, ex)
    }
}
```

先从操作数栈中弹出异常对象引用，如果该引用是 null，则抛出 NullPointerException 异常，否则看是否可以找到并跳转到异常处理代码。findAndGotoExceptionHandler() 函数的代码如下：

```
func findAndGotoExceptionHandler(thread *rtda.Thread, ex *heap.Object) bool {
    for {
        frame := thread.CurrentFrame()
        pc := frame.NextPC() - 1

        handlerPC := frame.Method().FindExceptionHandler(ex.Class(), pc)
        if handlerPC > 0 {
            stack := frame.OperandStack()
            stack.Clear()
            stack.PushRef(ex)
            frame.SetNextPC(handlerPC)
            return true
        }

        thread.PopFrame()
        if thread.IsStackEmpty() {
            break
        }
    }
    return false
}
```

从当前帧开始，遍历 Java 虚拟机栈，查找方法的异常处理表。假设遍历到帧 F，如果在 F 对应的方法中找不到异常处理项，则把 F 弹出，继续遍历。反之如果找到了异常处理项，在跳转到异常处理代码之前，要先把 F 的操作数栈清空，然后把异常对象引用推入栈

顶。OperandStack 结构体的 Clear() 方法是新增加的，后面给出它的代码。

如果遍历完 Java 虚拟机栈还是找不到异常处理代码，则 handleUncaughtException() 函数打印出 Java 虚拟机栈信息，代码如下：

```
func handleUncaughtException(thread *rtda.Thread, ex *heap.Object) {
    thread.ClearStack()

    jMsg := ex.GetRefVar("detailMessage", "Ljava/lang/String;")
    goMsg := heap.GoString(jMsg)
    println(ex.Class().JavaName() + ": " + goMsg)

    stes := reflect.ValueOf(ex.Extra())
    for i := 0; i < stes.Len(); i++ {
        ste := stes.Index(i).Interface().(interface {
            String() string
        })
        println("\tat " + ste.String())
    }
}
```

handleUncaughtException() 函数把 Java 虚拟机栈清空，然后打印出异常信息。由于 Java 虚拟机栈已经空了，所以解释器也就终止执行了。上面的代码使用 Go 语言的 reflect 包打印 Java 虚拟机栈信息。可以猜到，异常对象的 extra 字段中存放的就是 Java 虚拟机栈信息，那么这个 extra 字段是什么时候设置的呢？ 10.5 节会揭晓答案。前面的代码中还有几个方法没有介绍，现在依次给出它们的代码。

OperandStack 结构体的 Clear() 方法如下：

```
func (self *OperandStack) Clear() {
    self.size = 0
    for i := range self.slots {
        self.slots[i].ref = nil
    }
}
```

Thread 结构体的 ClearStack() 方法如下：

```
func (self *Thread) ClearStack() {
    self.stack.clear()
}
```

它调用了 Stack 结构体的 clear() 方法，代码如下：

```
func (self *Stack) clear() {
```

```
    for !self.isEmpty() {
        self.pop()
    }
}
```

athrow 指令实现后，还需要修改 ch10\instructions\factory.go 文件，在 NewInstruction() 函数中增加 athrow 指令的 case 语句，为了节约篇幅这里就不给出代码了。

10.5 Java 虚拟机栈信息

回到 ch10\native\java\lang\Throwable.go 文件，在其中定义 StackTraceElement 结构体，代码如下：

```
type StackTraceElement struct {
    fileName        string
    className       string
    methodName      string
    lineNumber      int
}
```

StackTraceElement 结构体用来记录 Java 虚拟机栈帧信息：lineNumber 字段给出帧正在执行哪行代码；methodName 字段给出方法名；className 字段给出声明方法的类名；fileName 字段给出类所在的文件名。下面实现 java.lang.Throwable 的 fillInStackTrace() 本地方法，代码如下：

```
// private native Throwable fillInStackTrace(int dummy);
func fillInStackTrace(frame *rtda.Frame) {
    this := frame.LocalVars().GetThis()
    frame.OperandStack().PushRef(this)

    stes := createStackTraceElements(this, frame.Thread())
    this.SetExtra(stes)
}
```

重点在 createStackTraceElements() 函数里，代码如下：

```
func createStackTraceElements(tObj *heap.Object, thread *rtda.Thread)
        []*StackTraceElement {
    skip := distanceToObject(tObj.Class()) + 2
    frames := thread.GetFrames()[skip:]
    stes := make([]*StackTraceElement, len(frames))
    for i, frame := range frames {
        stes[i] = createStackTraceElement(frame)
```

```
    }
    return stes
}
```

这个函数需要解释一下。由于栈顶两帧正在执行 fillInStackTrace(int) 和 fillInStackTrace() 方法，所以需要跳过这两帧。这两帧下面的几帧正在执行异常类的构造函数，所以也要跳过，具体要跳过多少帧数则要看异常类的继承层次。distanceToObject() 函数计算所需跳过的帧数，代码如下：

```
func distanceToObject(class *heap.Class) int {
    distance := 0
    for c := class.SuperClass(); c != nil; c = c.SuperClass() {
        distance++
    }
    return distance
}
```

计算好需要跳过的帧之后，调用 Thread 结构体的 GetFrames() 方法拿到完整的 Java 虚拟机栈，然后 reslice 一下就是真正需要的帧。GetFrames() 方法只是调用了 Stack 结构体的 getFrames() 方法，代码如下：

```
func (self *Thread) GetFrames() []*Frame {
    return self.stack.getFrames()
}
```

下面是 getFrames() 方法的代码。

```
func (self *Stack) getFrames() []*Frame {
    frames := make([]*Frame, 0, self.size)
    for frame := self._top; frame != nil; frame = frame.lower {
        frames = append(frames, frame)
    }
    return frames
}
```

createStackTraceElement() 函数根据帧创建 StackTraceElement 实例，代码如下：

```
func createStackTraceElement(frame *rtda.Frame) *StackTraceElement {
    method := frame.Method()
    class := method.Class()
    return &StackTraceElement{
        fileName:   class.SourceFile(),
        className:  class.JavaName(),
        methodName: method.Name(),
        lineNumber: method.GetLineNumber(frame.NextPC() - 1),
```

```
    }
}
```

最后实现 Class 结构体的 SourceFile() 方法和 Method 结构体的 GetLineNumber() 方法。打开 class.go，给 Class 结构体添加 sourceFile 字段，代码如下：

```
type Class struct {
    ... // 其他字段
    sourceFile string
}
```

SourceFile() 是 getter 方法，这里就不给出代码了。接下来需要修改 newClass() 函数，从 class 文件中读取源文件名，改动如下：

```
func newClass(cf *classfile.ClassFile) *Class {
    class := &Class{}
    ... // 其他代码
    class.sourceFile = getSourceFile(cf)
    return class
}
```

在 3.4.3 节讨论过，源文件名在 ClassFile 结构的属性表中，getSourceFile() 函数提取这个信息，代码如下：

```
func getSourceFile(cf *classfile.ClassFile) string {
    if sfAttr := cf.SourceFileAttribute(); sfAttr != nil {
        return sfAttr.FileName()
    }
    return "Unknown"
}
```

注意，并不是每个 class 文件中都有源文件信息，这个因编译时的编译器选项而异。Class 结构体改完了，下面修改 Method 结构体。打开 method.go，给 Method 结构体添加 lineNumberTable 字段，改动如下：

```
type Method struct {
    ... // 其他字段
    lineNumberTable *classfile.LineNumberTableAttribute
}
```

然后修改 copyAttributes() 方法，从 class 文件中提取行号表，代码如下：

```
func (self *Method) copyAttributes(cfMethod *classfile.MemberInfo) {
    if codeAttr := cfMethod.CodeAttribute(); codeAttr != nil {
        self.maxStack = codeAttr.MaxStack()
```

```
        self.maxLocals = codeAttr.MaxLocals()
        self.code = codeAttr.Code()
        self.lineNumberTable = codeAttr.LineNumberTableAttribute()
        self.exceptionTable = newExceptionTable(codeAttr.ExceptionTable(),
            self.class.constantPool)
    }
}
```

最后添加 GetLineNumber() 方法，代码如下：

```
func (self *Method) GetLineNumber(pc int) int {
    if self.IsNative() {
        return -2
    }
    if self.lineNumberTable == nil {
        return -1
    }
    return self.lineNumberTable.GetLineNumber(pc)
}
```

和源文件名一样，并不是每个方法都有行号表。如果方法没有行号表，自然也就查不到 pc 对应的行号，这种情况下返回 –1。本地方法没有字节码，这种情况下返回 –2。剩下的情况调用 LineNumberTableAttribute 结构体的 GetLineNumber() 方法查找行号，代码如下：

```
func (self *LineNumberTableAttribute) GetLineNumber(pc int) int {
    for i := len(self.lineNumberTable) - 1; i >= 0; i-- {
        entry := self.lineNumberTable[i]
        if pc >= int(entry.startPc) {
            return int(entry.lineNumber)
        }
    }
    return -1
}
```

上面的代码在 classfile\attr_line_number_table.go 文件中，行号表的更多信息请参考 3.4.7 节。

10.6 测试本章代码

打开命令行窗口，执行下面的命令编译本章代码。

```
go install jvmgo\ch10
```

命令执行完毕后，在 D:\go\workspace\bin 目录下出现 ch10.exe 文件。运行 ch10.exe，

测试下面的 Java 程序。

```
package jvmgo.book.ch10;

public class ParseIntTest {
    public static void main(String[] args) {
        foo(args);
    }
    private static void foo(String[] args) {
        try {
            bar(args);
        } catch (NumberFormatException e) {
            System.out.println(e.getMessage());
        }
    }
    private static void bar(String[] args) {
        if (args.length == 0) {
            throw new IndexOutOfBoundsException("no args!");
        }
        int x = Integer.parseInt(args[0]);
        System.out.println(x);
    }
}
```

笔者使用不同的参数进行测试，结果如图 10-1 所示。

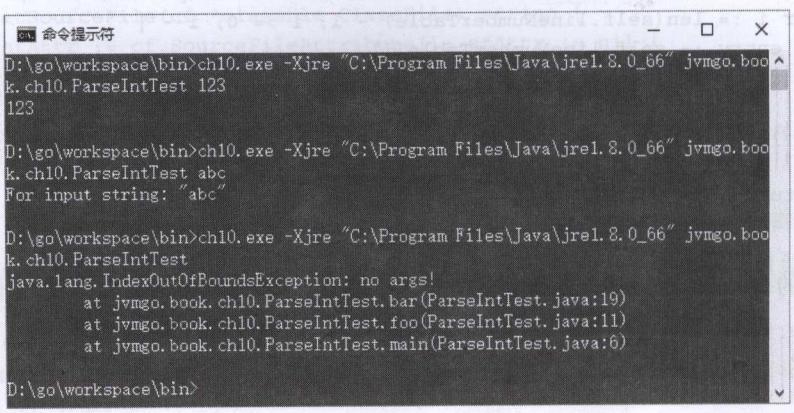
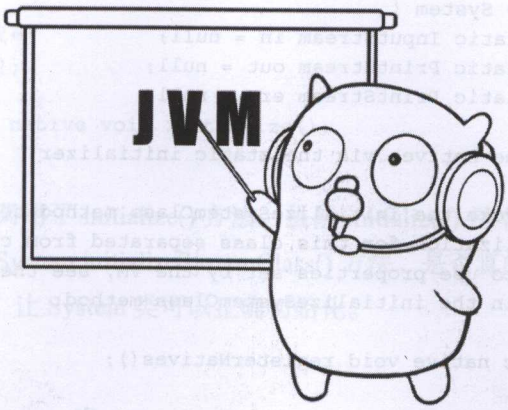


图 10-1 ParseIntTest 程序测试结果

10.7 本章小结

本章讨论了异常抛出和处理、异常处理表、athrow 指令等，下一章将讨论类加载器。



第11章 结 束

在第7章讨论了方法调用和返回，在第8章讨论了数组和字符串。经过整整8章的努力之后，“Hello，World！”终于出现在了控制台上。不过比较遗憾的是，由于java.lang.System类还没有被正确初始化，直接调用System.out.println()方法会导致NullPointerException异常抛出。为此修改了invokevirtual指令，对println()方法做了特殊处理。本章将弥补这个遗憾，把这个hack从代码中删除，让Java虚拟机可以真正在控制台上打印数字和字符串。

本章也是本书的最后一章，在结尾会对全书内容进行简要回顾。开始本章之前，还是先把目录结构准备好。复制ch10目录，改名为ch11。修改main.go等文件，把import语句中的ch10全都替换成ch11。本章对目录结构没有太大的调整。

11.1 System类是如何被初始化的

大家都知道，System类有3个公开的静态常量：out、err和in。其中out和err用于向标准输出流和标准错误流中写入信息，in用于从标准输入流中读取信息。那么这3个常量是在哪里被赋值的呢？看一下System类的源代码：

```
// java.lang.System
```

```
public final class System {
    public final static InputStream in = null;
    public final static PrintStream out = null;
    public final static PrintStream err = null;

    /* register the natives via the static initializer.
     *
     * VM will invoke the initializeSystemClass method to complete
     * the initialization for this class separated from clinit.
     * Note that to use properties set by the VM, see the constraints
     * described in the initializeSystemClass method.
     */
    private static native void registerNatives();
    static {
        registerNatives();
    }
    ... // 其他代码
}
```

从注释可知，System 类的初始化过程分为两个阶段。第一个阶段由类初始化方法完成，在这个方法中 registerNatives() 方法会注册其他本地方法。第二个阶段由 VM 完成，在这个阶段 VM 会调用 System.initializeSystemClass() 方法。那么 initializeSystemClass() 方法究竟干了些什么呢？这个方法很长，而且有很详细的注释。去掉与本节讨论无关的代码和注释之后，它的代码如下：

```
/**
 * Initialize the system class.  Called after thread initialization.
 */
private static void initializeSystemClass() {
    ... // 其他代码
    FileInputStream fdIn = new FileInputStream(FileDescriptor.in);
    FileOutputStream fdOut = new FileOutputStream(FileDescriptor.out);
    FileOutputStream fdErr = new FileOutputStream(FileDescriptor.err);
    setIn0(new BufferedInputStream(fdIn));
    setOut0(newPrintStream(fdOut, props.getProperty("sun.stdout.encoding")));
    setErr0(newPrintStream(fdErr, props.getProperty("sun.stderr.encoding")));
    ... // 其他代码
}
```

可见 in、out 和 err 正是在这里设置的。再来看 sun.misc.VM 类的源代码（VM 类属于 Oracle 私有代码，并没有开源，下面是反编译后的 Java 代码）：

```
// sun.misc.VM
public class VM {
    ... // 其他代码
```

```
    static {
        ... // 其他代码
        initialize();
    }
    private static native void initialize();
}
```

VM类在初始化时调用了initialize()方法。虽然initialize()是本地方法，但是可以推测正是这个方法调用了System.initializeSystemClass()方法。是否真的是这样笔者就不做考证了，下面修改解释器，让System类可以正确初始化。

11.2 初始化System类

先打开ch11\instructions\references\invokevirtual.go文件，修改invokevirtual指令的Execute()方法，把其中的hack代码删掉。由于只是删除代码，这里就不做详细说明了。

接下来打开ch11\native\sun\misc\VM.go文件，删除heap包的导入语句。原来的initialize()方法也是用hack方式实现的，需要重写，代码如下：

```
// private static native void initialize();
func initialize(frame *rtda.Frame) {
    classLoader := frame.Method().Class().Loader()
    jlSysClass := classLoader.LoadClass("java/lang/System")
    initSysClass := jlSysClass.GetStaticMethod("initializeSystemClass", "()V")
    base.InvokeMethod(frame, initSysClass)
}
```

新的实现只是调用了System.initializeSystemClass()方法而已。下面修改解释器，让它在执行主类的main()方法之前先调用VM. initialize()方法。为了让代码的可读性更好，将对main.go文件进行比较大的调整。打开ch11\main.go，把下面的代码复制进去：

```
package main

func main() {
    cmd := parseCmd()
    if cmd.versionFlag {
        println("version 0.0.1")
    } else if cmd.helpFlag || cmd.class == "" {
        printUsage()
    } else {
        newJVM(cmd).start()
    }
```

```
}
```

主要逻辑都被挪到了（新增加的）ch11\jvm.go 文件中，代码如下：

```
package main

import "fmt"
import "strings"
import "jvmgo/ch11/classpath"
import "jvmgo/ch11/instructions/base"
import "jvmgo/ch11/rtda"
import "jvmgo/ch11/rtda/heap"

type JVM struct {
    cmd             *Cmd
    classLoader     *heap.ClassLoader
    mainThread      *rtda.Thread
}

func newJVM(cmd *Cmd) *JVM {...}
func (self *JVM) start() {...}
```

newJVM() 函数创建 JVM 结构体实例，代码如下：

```
func newJVM(cmd *Cmd) *JVM {
    cp := classpath.Parse(cmd.XjreOption, cmd.cpOption)
    classLoader := heap.NewClassLoader(cp, cmd.verboseClassFlag)
    return &JVM{
        cmd:         cmd,
        classLoader: classLoader,
        mainThread:  rtda.NewThread(),
    }
}
```

start() 方法先初始化 VM 类，然后执行主类的 main() 方法，代码如下：

```
func (self *JVM) start() {
    self.initVM()
    self.execMain()
}
```

initVM() 先加载 sun.mis.VM 类，然后执行其类初始化方法，代码如下：

```
func (self *JVM) initVM() {
    vmClass := self.classLoader.LoadClass("sun/misc/VM")
    base.InitClass(self.mainThread, vmClass)
    interpret(self.mainThread, self.cmd.verboseInstFlag)
}
```

execMain() 方法先加载主类，然后执行其 main() 方法，代码如下：

```
func (self *JVM) execMain() {
    className := strings.Replace(self.cmd.class, ".", "/", -1)
    mainClass := self.classLoader.LoadClass(className)
    mainMethod := mainClass.GetMainMethod()
    if mainMethod == nil {
        fmt.Printf("Main method not found in class %s\n", self.cmd.class)
        return
    }

    argsArr := self.createArgsArray()
    frame := self.mainThread.NewFrame(mainMethod)
    frame.LocalVars().SetRef(0, argsArr) // 给 main() 方法传递 args 参数
    self.mainThread.PushFrame(frame)
    interpret(self.mainThread, self.cmd.verboseInstFlag)
}
```

execMain() 方法的前半部分代码是从 main.go 文件中拷贝过来的，我们已经比较熟悉了。后半部分代码需要解释的一点是：在调用 main() 方法之前，需要给它传递 args 参数，这是通过直接操作局部变量表实现的。createArgsArray() 方法把 Go 的 []string 变量转换成 Java 的字符串数组，代码是从 interpreter.go 文件中拷贝过来的，如下所示：

```
func (self *JVM) createArgsArray() *heap.Object {
    stringClass := self.classLoader.LoadClass("java/lang/String")
    argsLen := uint(len(self.cmd.args))
    argsArr := stringClass.ArrayClass().NewArray(argsLen)
    jArgs := argsArr.Refs()
    for i, arg := range self.cmd.args {
        jArgs[i] = heap.JString(self.classLoader, arg)
    }
    return argsArr
}
```

jvm.go 文件改好了，下面修改 interpret() 函数。打开 ch11\interpreter.go，删除 heap 包的导入语句和 createArgsArray() 函数，然后修改 interpret() 函数，代码如下：

```
func interpret(thread *rtda.Thread, logInst bool) {
    defer catchErr(thread)
    loop(thread, logInst)
}
```

修改之后 interpret() 函数简单了许多，直接调用 loop() 函数进入循环即可。至此，解释器修改完毕。这就是本章要写的全部代码吗？并不是。为了正常执行 System.initializeSystemClass() 以及 System.out.println() 等方法，还需要实现很多 Java 类库中的本

地方法。为了节约篇幅，这里就不一一列举了，请读者阅读随书源代码。下面以 System.out.println(String) 为例解释字符串是如何被打印到控制台的，其他类型变量的打印原理同字符串类似。

11.3 System.out.println() 是如何工作的

回到 System.initializeSystemClass() 方法，进一步省略之后，其代码如下：

```
// java.lang.System
public final static PrintStream out = null;
private static void initializeSystemClass() {
   ... // 其他代码
   FileOutputStream fdOut = new FileOutputStream(FileDescriptor.out);
   setOut0(newPrintStream(fdOut, props.getProperty("sun.stdout.encoding")));
   ... // 其他代码
}
```

setOut0() 是个本地方法，代码如下：

```
private static native void setOut0(PrintStream out);
```

newPrintStream() 方法的代码如下：

```
private static PrintStream newPrintStream(FileOutputStream fos, String enc) {
  if (enc != null) {
      try {
         return new PrintStream(new BufferedOutputStream(fos, 128), true, enc);
      } catch (UnsupportedEncodingException uee) {}
   }
   return new PrintStream(new BufferedOutputStream(fos, 128), true);
}
```

由代码可知，System.out 常量是 PrintStream 类型，它内部包装了一个 BufferedOutputStream 实例。BufferedOutputStream 内部又包装了一个 FileOutputStream 实例。Java 的 io 类库使用了装饰器模式，调用 System.out.println(String) 方法之后，经过层层包装，最后到达 FileOutputStream 类的 writeBytes() 方法。这个方法无法用 Java 代码实现，所以是个本地方法，其代码如下：

```
// java.io.FileOutputStream
public class FileOutputStream extends OutputStream {
   ... // 其他代码
   private native void writeBytes(byte b[], int off, int len, boolean append)
```

```
        throws IOException;
}
```

System.setOut0() 本地方法在 ch11\native\java\lang\System.go 文件中实现，代码如下：

```
// private static native void setOut0(PrintStream out);
func setOut0(frame *rtda.Frame) {
    out := frame.LocalVars().GetRef(0)
    sysClass := frame.Method().Class()
    sysClass.SetRefVar("out", "Ljava/io/PrintStream;", out)
}
```

FileOutputStream.writeBytes() 本地方法在 ch11\native\java\io\FileOutputStream.go 文件中实现，代码如下：

```
// private native void writeBytes(byte b[], int off, int len, boolean append)
// throws IOException;
func writeBytes(frame *rtda.Frame) {
    vars := frame.LocalVars()
    //this := vars.GetRef(0)
    b := vars.GetRef(1)
    off := vars.GetInt(2)
    len := vars.GetInt(3)
    //append := vars.GetBoolean(4)

    jBytes := b.Data().([]int8)
    goBytes := castInt8sToUint8s(jBytes)
    goBytes = goBytes[off : off+len]
    os.Stdout.Write(goBytes)
}
```

虽然同是字节类型，但是在 Java 语言中 byte 是有符号类型，在 Go 语言中 byte 则是无符号类型。所以这里需要先把 Java 的字节数组转换成 Go 的 []byte 变量，然后再调用 os.Stdout.Write() 方法把它写到控制台。castInt8sToUint8s() 函数代码如下：

```
func castInt8sToUint8s(jBytes []int8) (goBytes []byte) {
    ptr := unsafe.Pointer(&jBytes)
    goBytes = *((*[]byte)(ptr))
    return
}
```

如果读者属于完美主义者，很容易会发现这里的小瑕疵：FileOutputStream 应该可以处理任何文件而不仅仅是标准输出。没错，不过本章就到此为止了，感兴趣的读者可以继续完善代码，让 FileOutputStream 可以支持任何文件。下面测试本章代码。

11.4 测试本章代码

打开命令行窗口，执行下面的命令编译本章代码：

```
go install jvmgo\ch11
```

命令执行完毕后，在 D:\go\workspace\bin 目录下出现 ch11.exe 文件。用 ch11.exe 测试 HelloWorld 程序，结果如图 11-1 所示。

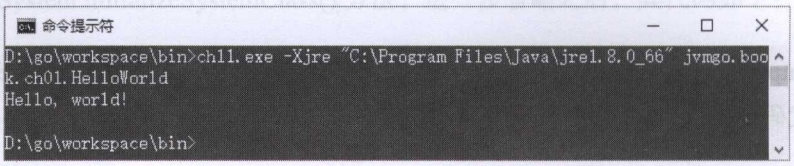


图 11-1 HelloWorld 程序执行结果

11.5 总结

本书共 11 章，各章内容如下：

第 1 章讨论了 Java 虚拟机是如何启动的，介绍了 java 命令的用法，并且实现了一个类似的命令行工具。

第 2、第 3、第 6、第 8 章讨论了类加载器。其中第 2 章讨论了 Java 虚拟机如何搜索 class 文件，并且实现了类路径，可以把 class 文件读到内存中。第 3 章讨论了 class 文件结构体，并且实现了 class 文件解析，把难以理解的字节序列转换成了 ClassFile 结构体。第 6 章实现了一个简化版的类加载器，进一步处理 ClassFile 结构体，把它转换成 Class 结构体放入方法区。第 8 章对类加载器进行了扩展，使其可以加载数组类。

第 4、第 6 章讨论了运行时数据区。其中第 4 章主要讨论了线程私有的运行时数据区，包括 Java 虚拟机栈、帧、局部变量表和操作数栈等。第 6 章主要讨论了线程共享的运行时数据区，包括方法区和运行时常量池等。

第 7、第 9、第 10 章讨论了方法调用。其中第 7 章主要讨论了 Java 方法的调用和返回，并且实现了相关指令。第 9 章讨论了本地方法调用，并且实现了 Java 类库中一些最重要的本地方法。第 10 章讨论了异常处理，并且实现了 athrow 指令。

第 5 章编写了一个简单的解释器，从这一章开始，我们陆续实现了约 200 条指令。

在Java虚拟机规范已经定义的205条指令中，只剩下8条还没有实现，分别是：控制指令中的jsr和ret；扩展指令中的jsr_w；引用类指令中的invokedynamic、monitorenter和monitorexit；以及保留指令中的breakpoint和impldep2。

不过遗憾的是，有很多重要的内容没有讨论：class文件验证、内存管理和垃圾回收、类加载器的委派模型、多线程、JIT，等等。如果本书有机会出第2版，希望可以涵盖这些内容。

附录 指 令 表

Constants

操作码	助记符	章节	操作码	助记符	章节
0x00	nop	5.3.1	0x0b	fconst_0	5.3.2
0x01	aconst_null	5.3.2	0x0c	fconst_1	5.3.2
0x02	iconst_m1	5.3.2	0x0d	fconst_2	5.3.2
0x03	iconst_0	5.3.2	0x0e	dconst_0	5.3.2
0x04	iconst_1	5.3.2	0x0f	dconst_1	5.3.2
0x05	iconst_2	5.3.2	0x10	bipush	5.3.3
0x06	iconst_3	5.3.2	0x11	sipush	5.3.3
0x07	iconst_4	5.3.2	0x12	ldc	6.6.5
0x08	iconst_5	5.3.2	0x13	ldc_w	8.5.2
0x09	lconst_0	5.3.2	0x14	ldc2_w	9.3.4
0x0a	lconst_1	5.3.2			

Loads

操作码	助记符	章节	操作码	助记符	章节
0x15	iload	5.4	0x26	dload_0	5.4
0x16	lload	5.4	0x27	dload_1	5.4
0x17	fload	5.4	0x28	dload_2	5.4
0x18	dload	5.4	0x29	dload_3	5.4
0x19	aload	5.4	0x2a	aload_0	5.4
0x1a	iload_0	5.4	0x2b	aload_1	5.4
0x1b	iload_1	5.4	0x2c	aload_2	5.4
0x1c	iload_2	5.4	0x2d	aload_3	5.4
0x1d	iload_3	5.4	0x2e	iaload	8.3.4
0x1e	lload_0	5.4	0x2f	laload	8.3.4
0x1f	lload_1	5.4	0x30	faload	8.3.4
0x20	lload_2	5.4	0x31	daload	8.3.4

（续）

操作码	助记符	章节	操作码	助记符	章节
0x21	lload_3	5.4	0x32	aaload	8.3.4
0x22	fload_0	5.4	0x33	baload	8.3.4
0x23	fload_1	5.4	0x34	caload	8.3.4
0x24	fload_2	5.4	0x35	saload	8.3.4
0x25	fload_3	5.4			

Stores

操作码	助记符	章节	操作码	助记符	章节
0x36	istore	5.5	0x47	dstore_0	5.5
0x37	lstore	5.5	0x48	dstore_1	5.5
0x38	fstore	5.5	0x49	dstore_2	5.5
0x39	dstore	5.5	0x4a	dstore_3	5.5
0x3a	astore	5.5	0x4b	astore_0	5.5
0x3b	istore_0	5.5	0x4c	astore_1	5.5
0x3c	istore_1	5.5	0x4d	astore_2	5.5
0x3d	istore_2	5.5	0x4e	astore_3	5.5
0x3e	istore_3	5.5	0x4f	iastore	8.3.5
0x3f	lstore_0	5.5	0x50	lastore	8.3.5
0x40	lstore_1	5.5	0x51	fastore	8.3.5
0x41	lstore_2	5.5	0x52	dastore	8.3.5
0x42	lstore_3	5.5	0x53	aastore	8.3.5
0x43	fstore_0	5.5	0x54	bastore	8.3.5
0x44	fstore_1	5.5	0x55	castore	8.3.5
0x45	fstore_2	5.5	0x56	sastore	8.3.5
0x46	fstore_3	5.5			

Stack

操作码	助记符	章节	操作码	助记符	章节
0x57	pop	5.6.1	0x5c	dup2	5.6.2
0x58	pop2	5.6.1	0x5d	dup2_x1	5.6.2
0x59	dup	5.6.2	0x5e	dup2_x2	5.6.2
0x5a	dup_x1	5.6.2	0x5f	swap	5.6.3
0x5b	dup_x2	5.6.2			

Math

操作码	助记符	章节	操作码	助记符	章节
0x60	iadd	5.7.1	0x74	ineg	5.7.1
0x61	ladd	5.7.1	0x75	lneg	5.7.1
0x62	fadd	5.7.1	0x76	fneg	5.7.1
0x63	dadd	5.7.1	0x77	dneg	5.7.1
0x64	isub	5.7.1	0x78	ishl	5.7.2
0x65	lsub	5.7.1	0x79	lshl	5.7.2
0x66	fsub	5.7.1	0x7a	ishr	5.7.2
0x67	dsub	5.7.1	0x7b	lshr	5.7.2
0x68	imul	5.7.1	0x7c	iushr	5.7.2
0x69	lmul	5.7.1	0x7d	lushr	5.7.2
0x6a	fmul	5.7.1	0x7e	iand	5.7.3
0x6b	dmul	5.7.1	0x7f	land	5.7.3
0x6c	idiv	5.7.1	0x80	ior	5.7.3
0x6d	ldiv	5.7.1	0x81	lor	5.7.3
0x6e	fdiv	5.7.1	0x82	ixor	5.7.3
0x6f	ddiv	5.7.1	0x83	lxor	5.7.3
0x70	irem	5.7.1	0x84	iinc	5.7.4
0x71	lrem	5.7.1			
0x72	frem	5.7.1			
0x73	drem	5.7.1			

Conversions

操作码	助记符	章节	操作码	助记符	章节
0x85	i2l	5.8	0x8e	d2i	5.8
0x86	i2f	5.8	0x8f	d2l	5.8
0x87	i2d	5.8	0x90	d2f	5.8
0x88	l2i	5.8	0x91	i2b	5.8
0x89	l2f	5.8	0x92	i2c	5.8
0x8a	l2d	5.8	0x93	i2s	5.8
0x8b	f2i	5.8			
0x8c	f2l	5.8			
0x8d	f2d	5.8			

Comparisons

操作码	助记符	章节	操作码	助记符	章节
0x94	lcmp	5.9.1	0x9f	if_icmpeq	5.9.4
0x95	fcmpl	5.9.2	0xa0	if_icmpne	5.9.4
0x96	fcmpg	5.9.2	0xa1	if_icmplt	5.9.4
0x97	dcmpl	5.9.2	0xa2	if_icmpge	5.9.4
0x98	dcmpg	5.9.2	0xa3	if_icmpgt	5.9.4
0x99	ifeq	5.9.3	0xa4	if_icmple	5.9.4
0x9a	ifne	5.9.3	0xa5	if_acmpeq	5.9.5
0x9b	iflt	5.9.3	0xa6	if_acmpne	5.9.5
0x9c	ifge	5.9.3			
0x9d	ifgt	5.9.3			
0x9e	ifle	5.9.3			

Control

操作码	助记符	章节	操作码	助记符	章节
0xa7	goto	5.10.1	0xac	ireturn	7.4
0xa8	jsr		0xad	lreturn	7.4
0xa9	ret		0xae	freturn	7.4
0xaa	tableswitch	5.10.2	0xaf	dreturn	7.4
0xab	lookupswitch	5.10.3	0xb0	areturn	7.4
			0xb1	return	7.4

References

操作码	助记符	章节	操作码	助记符	章节
0xb2	getstatic	6.6.2	0xbb	new	6.6.1
0xb3	putstatic	6.6.2	0xbc	newarray	8.3.1
0xb4	getfield	6.6.3	0xbd	anewarray	8.3.2
0xb5	putfield	6.6.3	0xbe	arraylength	8.3.3
0xb6	invokevirtual	7.5.3	0xbf	athrow	10.4
0xb7	invokespecial	7.5.2	0xc0	checkcast	6.6.4
0xb8	invokestatic	7.5.1	0xc1	instanceof	8.3.7
0xb9	invokeinterface	7.5.4	0xc2	monitorenter	
0xba	invokedynamic		0xc3	monitorexit	

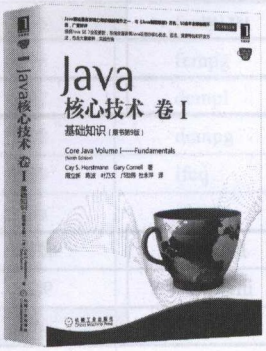
Extended

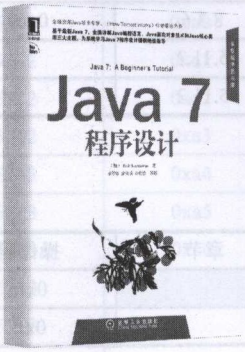
操作码	助记符	章节	操作码	助记符	章节
0xc4	wide	5.11.1	0xc8	goto_w	5.11.3
0xc5	multianewarray	8.3.6	0xc9	jsr_w	
0xc6	ifnull	5.11.2			
0xc7	ifnonnull	5.11.2			

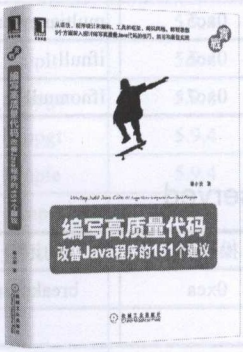
Reserved

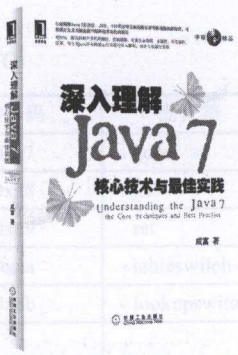
操作码	助记符	章节	操作码	助记符	章节
0xca	breakpoint		0xfe	impdep1	9.2
			0xff	impdep2	

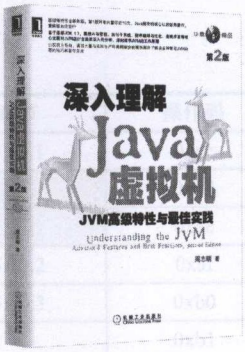
推荐阅读


Java
核心技术 卷I
基础知识（原书第9版）
Core Java Volume I—Fundamentals


Java 7
程序设计


编写高质量代码
改善Java程序的151个建议


深入理解
Java 7
核心技术与最佳实践


深入理解
Java
虚拟机
JVM高级特性与最佳实践


Java
应用架构设计
模块化模式与OSGi


Java安全编码标准


Java
并发编程实战
Java Concurrency in Practice


Java虚拟机并发编程

作者简介

张秀宏 资深Java服务器开发工程师，有多年的Java开发、游戏服务器开发和架构经验，对Java虚拟机有非常深入的研究。曾在EA、华娱无线等游戏公司担任高级服务器工程师职位，现任乐元素Lead软件工程师。

封面设计：陈子平

想要了解Java虚拟机的内部运行原理，阅读虚拟机规范、书籍、源码是一种常见的途径，而从零开始自己动手编写一个实验室性质的Java虚拟机，也许会是一种更加有趣且有效的学习路径。如果不考虑Java庞大类库的实现和JVM的实际生产力需求，仅是去“正确地”实现一台Java虚拟机，其实并不如大多数人所想的那样高深和困难——只需正确读取Class文件中每一条字节码指令，并且能正确执行这些指令所蕴含的操作即可。通过本书，您可以跟随作者的思路和指引，一步步完成Java虚拟机的各个组成部分，在动手的过程中了解Java虚拟机的运作原理。

—— **周志明**《深入理解Java虚拟机：JVM高级特性与最佳实践》作者

这是国内第一本以实战模式描述JVM原理的书！秀宏对JVM进行了大量研究，在书中深入浅出地分析了class文件的数据结构和JVM的基本原理，并使用Go语言用不到1万行的程序代码就实现了JVM的基本模型，是Java爱好者了解JVM实现原理的一本好书。实战才是最有效的掌握知识的手段，快快动手，实现属于自己的Java虚拟机吧！

—— **凌聪** 乐元素CTO

JVM对大多数的Java开发人员，无论是初出茅庐的菜鸟以及工作多年的老手，可能都还是一个神秘的、高深莫测的黑匣子。本书的出版，使作者通过一个个实践的方式，一步步带领大家饶有趣味地揭开JVM的神秘面纱，极大加深程序员对Java的理解，进而构建更加合理高效的代码。

—— **金智伟** 钱咸升（北京）网络科技股份公司CTO

上架指导：计算机 / 程序设计 / Java

ISBN 978-7-111-53413-6

9 787111 534136 >

定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn